Acknowledgments

The material Contained herein is not a license, either expressly or impliedly, to any intellectual property owned or Controlled By any of the authors or developers of this material or to any Contribution thereto. The material Contained herein is provided on an "AS IS" Basis and, to the maximum extent permitted By applicable law, this information is provided AS IS AND WITH ALL FAULTS, and the authors and developers of this material hereby disclaim all other warranties and Conditions, either express, implied or statutory, including, But not limited to, any (if any) implied warranties, duties or Conditions of merchantability, of fitness for a particular purpose, of accuracy or Completeness of responses, of results, of workmanlike effort, of lack of viruses and of lack of negligence, all with regard to this material and any Contribution thereto. Designers must not rely on the absence or Characteristics of any features or instructions marked "reserved" or "undefined." The Unified EFI Forum, Inc. reserves any features or instructions so marked for future definition and shall have no responsibility whatsoever for Conflicts or incompatibilities arising from future Changes to them. ALSO, THERE IS NO WARRANTY OR CONDITION OF TITLE, QUIET ENJOYMENT, QUIET POSSESSION, CORRESPONDENCE TO DESCRIPTION OR NON-INFRINGEMENT WITH REGARD TO THE SPECIFICATION AND ANY CONTRIBUTION THERETO.

IN NO EVENT WILL ANY AUTHOR OR DEVELOPER OF THIS MATERIAL OR ANY CONTRIBUTION THERETO BE LIABLE TO ANY OTHER PARTY FOR THE COST OF PROCURING SUBSTITUTE GOODS OR SERVICES, LOST PROFITS, LOSS OF USE, LOSS OF DATA, OR ANY INCIDENTAL, CONSEQUENTIAL, DIRECT, INDIRECT, OR SPECIAL DAMAGES WHETHER UNDER CONTRACT, TORT, WARRANTY, OR OTHERWISE, ARISING IN ANY WAY OUT OF THIS OR ANY OTHER AGREEMENT RELATING TO THIS DOCUMENT, WHETHER OR NOT SUCH PARTY HAD ADVANCE NOTICE OF THE POSSIBILITY OF SUCH DAMAGES.

Copyright © 2021, Unified Extensible Firmware Interface (UEFI) Forum, Inc. All Rights Reserved. The UEFI Forum is the owner of all rights and title in and to this work, including all copyright rights that may exist, and all rights to use and reproduce this work. Further to such rights, permission is hereby granted to any person implementing this specification to maintain an electronic version of this work accessible by its internal personnel, and to print a copy of this specification in hard copy form, in whole or in part, in each case solely for use by that person in connection with the implementation of this Specification, provided no modification is made to the Specification.
### Revision History

<table>
<thead>
<tr>
<th>Revision</th>
<th>Mantis Number / Description</th>
<th>Date</th>
</tr>
</thead>
<tbody>
<tr>
<td>2.9</td>
<td>1866 GetInfo() of Adapter Information Protocol should have a provision for IHV to return no data</td>
<td>March 2021</td>
</tr>
<tr>
<td>2.9</td>
<td>1982 Clarify the PKCS#7 SignedData structure of EFI_VARIABLE_AUTHENTICATION</td>
<td>March 2021</td>
</tr>
<tr>
<td>2.9</td>
<td>1986 Need a mechanism using which browser to exit out of IHV formset silently without any popup</td>
<td>March 2021</td>
</tr>
<tr>
<td>2.9</td>
<td>1989 NVDIMM SPA Location Cookie</td>
<td>March 2021</td>
</tr>
<tr>
<td>2.9</td>
<td>2024 CXL CPER Records</td>
<td>March 2021</td>
</tr>
<tr>
<td>2.9</td>
<td>2042 New Event Group EFI_EVENT_GROUP_AFTER_READY_TO_BOOT</td>
<td>March 2021</td>
</tr>
<tr>
<td>2.9</td>
<td>2043 New Event Group EFI_EVENT_GROUP_BEFORE_EXIT_BOOT_SERVICES</td>
<td>March 2021</td>
</tr>
<tr>
<td>2.9</td>
<td>2046 Add support for Key 14 &amp; 56 for Japanese keyboard layout</td>
<td>March 2021</td>
</tr>
<tr>
<td>2.9</td>
<td>2053 Figure/Table Numbers are Duplicated in Appendices</td>
<td>March 2021</td>
</tr>
<tr>
<td>2.9</td>
<td>2062 Table numbering to restart for each chapter</td>
<td>March 2021</td>
</tr>
<tr>
<td>2.9</td>
<td>2065 CXL proposal for CDAT table extraction from devices</td>
<td>March 2021</td>
</tr>
<tr>
<td>2.9</td>
<td>2093 UpdateCapsule ScatterGatherList cache maintenance</td>
<td>March 2021</td>
</tr>
<tr>
<td>2.9</td>
<td>2129 Add DTB Configuration Table standard GUID</td>
<td>March 2021</td>
</tr>
<tr>
<td>2.9</td>
<td>2131 Clarify Console requirements</td>
<td>March 2021</td>
</tr>
<tr>
<td>2.9</td>
<td>2134 Introduce unacceptable memory type</td>
<td>March 2021</td>
</tr>
<tr>
<td>2.9</td>
<td>2155 Typo in Arm Processor CPER Error Section</td>
<td>March 2021</td>
</tr>
<tr>
<td>2.9</td>
<td>2167 CPER for CXL Component Events</td>
<td>March 2021</td>
</tr>
<tr>
<td>2.9</td>
<td>2185 Declaration for UEFI 2.9 specification in the System Table</td>
<td>March 2021</td>
</tr>
<tr>
<td>2.9</td>
<td>2190 Misc. spec review feedback</td>
<td>March 2021</td>
</tr>
<tr>
<td>2.9</td>
<td>2199 EFI_IMAGE_EXECUTION_INFO_TABLE references</td>
<td>March 2021</td>
</tr>
<tr>
<td>2.9</td>
<td>2200 Config tables references from section 4.6</td>
<td>March 2021</td>
</tr>
<tr>
<td>2.9</td>
<td>2204 Typo in GUID definition for EFI_MANAGED_NETWORK_SERVICE_BINDING_PROTOCOL</td>
<td>March 2021</td>
</tr>
<tr>
<td>2.9</td>
<td>2212 Incorrect cross reference to User Information Table</td>
<td>March 2021</td>
</tr>
<tr>
<td>2.8 C</td>
<td>2117 - EFI_BROWSER_ACTION_REQUEST_RECONNECT - perform the action when user exits out of formset</td>
<td>Jan. 2021</td>
</tr>
<tr>
<td>2.8 C</td>
<td>2139 Update RISC-V UEFI corresponding spec to align with latest RISC-V spec</td>
<td>Jan. 2021</td>
</tr>
<tr>
<td>2.8 C</td>
<td>2155 Typo in Arm Processor CPER Error Section</td>
<td>Jan. 2021</td>
</tr>
<tr>
<td>2.8 C</td>
<td>2158 EFI_DRIVER_HEALTH_PROTOCOL.GetHealthStatus() - Driver not managing any controller</td>
<td>Jan. 2021</td>
</tr>
<tr>
<td>2.8 C</td>
<td>2172 Revise EFI_REDFISH_DISCOVER_PROTOCOL definitions to match the implementation.</td>
<td>Jan. 2021</td>
</tr>
<tr>
<td>2.8 C</td>
<td>2173 Question on EFI_CAPSULE_HEADER Flags definition</td>
<td>Jan. 2021</td>
</tr>
<tr>
<td>2.8 C</td>
<td>2184 FI_BOOT_MANAGER_POLICY_PROTOCOL typos</td>
<td>Jan. 2021</td>
</tr>
<tr>
<td>2.8 C</td>
<td>2190 EFI_SUCCESS misspelled in five places</td>
<td>Jan. 2021</td>
</tr>
<tr>
<td>Revision</td>
<td>Mantis Number / Description</td>
<td>Date</td>
</tr>
<tr>
<td>----------</td>
<td>-----------------------------</td>
<td>------------</td>
</tr>
<tr>
<td>2.8 B</td>
<td>1926 update: corrected EFI_SYSTEM_TABLE entries from 2_8 to 2_80</td>
<td>May 2020</td>
</tr>
<tr>
<td>2.8 B</td>
<td>1935 update: removed a space from several references to the EFI_JSON_CAPSULE_ID_GUID</td>
<td>May 2020</td>
</tr>
<tr>
<td>2.8 B</td>
<td>2073 Modify definition of &quot;DPA&quot; for use with CXL based devices</td>
<td>May 2020</td>
</tr>
<tr>
<td>2.8 B</td>
<td>2074 Memory Range typo</td>
<td>May 2020</td>
</tr>
<tr>
<td>2.8 B</td>
<td>2080 Typo in N.2.2 Section Descriptor Table 56</td>
<td>May 2020</td>
</tr>
<tr>
<td>2.8 B</td>
<td>2083 Typo in guid definition of CCIX PER Log Error Section</td>
<td>May 2020</td>
</tr>
<tr>
<td>2.8 B</td>
<td>2088 Clarifications on caller-freed buffers</td>
<td>May 2020</td>
</tr>
<tr>
<td>2.8 B</td>
<td>2091 Inconsistency in description of EFI_FIRMWARE_MANAGEMENT_CAPSULE_IMAGE_HEADER structure</td>
<td>May 2020</td>
</tr>
<tr>
<td>2.8 B</td>
<td>2092 Typo in definition of PEI Notification type in Table 269. Error record header</td>
<td>May 2020</td>
</tr>
<tr>
<td>2.8 B</td>
<td>2095 PCI I/O attribute typos in section 14.4 &quot;EFI PCI I/O Protocol&quot;</td>
<td>May 2020</td>
</tr>
<tr>
<td>2.8 B</td>
<td>UEFI Runtime Service Table correction</td>
<td>May 2020</td>
</tr>
<tr>
<td>2.8 B</td>
<td>2096 Typo in definition of EFI_JSON_CONFIG_DATA_ITEM</td>
<td>May 2020</td>
</tr>
<tr>
<td>2.8 A</td>
<td>1970 Security Command Protocol change for OPAL RAID devices</td>
<td>February 2020</td>
</tr>
<tr>
<td>2.8 A</td>
<td>1998 Update RISC-V related spec</td>
<td>February 2020</td>
</tr>
<tr>
<td>2.8 A</td>
<td>2000 JSON Capsule clarification</td>
<td>February 2020</td>
</tr>
<tr>
<td>2.8 A</td>
<td>2002 Memory allocations between ExitBootServices calls</td>
<td>February 2020</td>
</tr>
<tr>
<td>2.8 A</td>
<td>2009 DMTF references in UEFI spec</td>
<td>February 2020</td>
</tr>
<tr>
<td>2.8 A</td>
<td>2013 Correct EFI_BOOT_SERVICES.DisconnectController contradicting info in Description</td>
<td>February 2020</td>
</tr>
<tr>
<td>2.8 A</td>
<td>2018 EFI_EDID_OVERRIDE_PROTOCOL_GET_EDID should take an EFI_HANDLE as ChildHandle</td>
<td>February 2020</td>
</tr>
<tr>
<td>2.8 A</td>
<td>2020 EFI_LOADED_IMAGE_PROTOCOL.LoadOptions does not mention it is related to Load Options.</td>
<td>February 2020</td>
</tr>
<tr>
<td>2.8 A</td>
<td>2025 Capsule Depex Length Declaration</td>
<td>February 2020</td>
</tr>
<tr>
<td>2.8 A</td>
<td>2026 FMP Capsule Image Header extension</td>
<td>February 2020</td>
</tr>
<tr>
<td>2.8 A</td>
<td>2029 Add missing GUIDs in Appendix N</td>
<td>February 2020</td>
</tr>
<tr>
<td>2.8 A</td>
<td>2030 Fix spec index to show the Appendix chapters</td>
<td>February 2020</td>
</tr>
<tr>
<td>2.8 A</td>
<td>2034 Depex added description</td>
<td>February 2020</td>
</tr>
<tr>
<td>2.8 A</td>
<td>2035 Fix OUT parameters marked as IN OUT</td>
<td>February 2020</td>
</tr>
<tr>
<td>2.8 A</td>
<td>2036 SetVariable errata: clarify that in-place variable update is supported</td>
<td>February 2020</td>
</tr>
<tr>
<td>2.8 A</td>
<td>2038 Configuration Tables Errata</td>
<td>February 2020</td>
</tr>
<tr>
<td>2.8 A</td>
<td>2041 EFI_EVENT_GROUP_EXIT_BOOT_SERVICES Errata</td>
<td>February 2020</td>
</tr>
<tr>
<td>2.8 A</td>
<td>2050 Incomplete list of EFI_SERVICE_BINDING_PROTOCOL protocols</td>
<td>February 2020</td>
</tr>
<tr>
<td>2.8 A</td>
<td>2049 RuntimeServicesSupported EFI variable should be a config table</td>
<td>February 2020</td>
</tr>
<tr>
<td>2.8 A</td>
<td>2051 Typo in Table - CPER IA32/X64 Bus Check Structure</td>
<td>February 2020</td>
</tr>
<tr>
<td>2.8 A</td>
<td>2053 Figure/Table Numbers are Duplicated in Appendices</td>
<td>February 2020</td>
</tr>
<tr>
<td>2.8</td>
<td>1832 Extend SERIAL_IO with DeviceTypeGuid</td>
<td>March 2019</td>
</tr>
<tr>
<td>2.8</td>
<td>1834 UEFI REST EX Protocol</td>
<td>March 2019</td>
</tr>
<tr>
<td>Revision</td>
<td>Mantis Number / Description</td>
<td>Date</td>
</tr>
<tr>
<td>----------</td>
<td>---------------------------------------------------------------------------------------------</td>
<td>------------</td>
</tr>
<tr>
<td>2.8</td>
<td>1853 Adding support for a REST style formset</td>
<td>March 2019</td>
</tr>
<tr>
<td>2.8</td>
<td>1858 New Device Path for bootable NVDIMM namespaces</td>
<td>March 2019</td>
</tr>
<tr>
<td>2.8</td>
<td>1861 New EFI_MEMORY_RANGE_CAPSULE Descriptor</td>
<td>March 2019</td>
</tr>
<tr>
<td>2.8</td>
<td>1866 GetInfo() of Adapter Information Protocol should have a provision for IHV to</td>
<td>March 2019</td>
</tr>
<tr>
<td></td>
<td>return no data</td>
<td>March 2019</td>
</tr>
<tr>
<td>2.8</td>
<td>1872 Peripheral-attached Memory</td>
<td>March 2019</td>
</tr>
<tr>
<td>2.8</td>
<td>1876 Remove the EBC support requirement</td>
<td>March 2019</td>
</tr>
<tr>
<td>2.8</td>
<td>1879 Clarification of REST (EX) protocol</td>
<td>March 2019</td>
</tr>
<tr>
<td>2.8</td>
<td>1908 Update of uncommitted data in the FOROM_OPEN callback</td>
<td>March 2019</td>
</tr>
<tr>
<td>2.8</td>
<td>1919 Memory Cryptography Attribute</td>
<td>March 2019</td>
</tr>
<tr>
<td>2.8</td>
<td>1920 Redfish Discover Protocol</td>
<td>March 2019</td>
</tr>
<tr>
<td>2.8</td>
<td>1921 HTTPS hostname validation</td>
<td>March 2019</td>
</tr>
<tr>
<td>2.8</td>
<td>1924 Update to EFI_REST_EX_PROTOCOL.AsyncSendReceive</td>
<td>March 2019</td>
</tr>
<tr>
<td>2.8</td>
<td>1925 Clarify requirement of REST related protocols</td>
<td>March 2019</td>
</tr>
<tr>
<td>2.8</td>
<td>1926 New UEFI Spec Revision --&gt; 2.8</td>
<td>March 2019</td>
</tr>
<tr>
<td>2.8</td>
<td>1935 UEFI JSON Capsule Support</td>
<td>March 2019</td>
</tr>
<tr>
<td>2.8</td>
<td>1936 ResetSystem - support ResetData for all status scenarios.</td>
<td>March 2019</td>
</tr>
<tr>
<td>2.8</td>
<td>1937 Behavior of default values</td>
<td>March 2019</td>
</tr>
<tr>
<td>2.8</td>
<td>1941 New EFI REST JSON Structure Protocol</td>
<td>March 2019</td>
</tr>
<tr>
<td>2.8</td>
<td>1942 Adding dependency expression capability into FMP type capsules</td>
<td>March 2019</td>
</tr>
<tr>
<td>2.8</td>
<td>1947 Keyword strings of Configuration Keyword Handler Protocol Enhancements</td>
<td>March 2019</td>
</tr>
<tr>
<td>2.8</td>
<td>1953 Add document version# conventions</td>
<td>March 2019</td>
</tr>
<tr>
<td>2.8</td>
<td>1954 set (*Attributes) when GetVariable() returns EFI_BUFFER_TOO_SMALL and Attributes is non-NULL</td>
<td>March 2019</td>
</tr>
<tr>
<td>2.8</td>
<td>1956 Platform to honor ActionRequest for Action changing</td>
<td>March 2019</td>
</tr>
<tr>
<td>2.8</td>
<td>1961 Add EFI_UNSUPPORTED to EFI_RUNTIME_SERVICES calls</td>
<td>March 2019</td>
</tr>
<tr>
<td>2.8</td>
<td>1966 Add new capsule processing error codes</td>
<td>March 2019</td>
</tr>
<tr>
<td>2.8</td>
<td>1974 Add new CCIX PER Log Error Section to appendix</td>
<td>March 2019</td>
</tr>
<tr>
<td>2.8</td>
<td>1996 Firmware Processing of the Capsule Identified by EFI_JSON_CAPSULE_ID_GUID</td>
<td>March 2019</td>
</tr>
<tr>
<td>2.7B</td>
<td>1773 Clarify The EFI System Table entry for capsule image</td>
<td>March 2019</td>
</tr>
<tr>
<td>2.7B</td>
<td>1801 ExtractConfig() format may change when called multiple times</td>
<td>March 2019</td>
</tr>
<tr>
<td>2.7B</td>
<td>1835 Misleading / unclear statement about EFI-bootability of UDF media</td>
<td>March 2019</td>
</tr>
<tr>
<td>2.7B</td>
<td>1838 RGB/BGR Contradiction in 2.7 GOP</td>
<td>March 2019</td>
</tr>
<tr>
<td>2.7B</td>
<td>1841 BluetoothLE ECR - support autoreconnect</td>
<td>March 2019</td>
</tr>
<tr>
<td>2.7B</td>
<td>1842 BluetoothLE ECR - Add missing ConnectionCompleteCallback</td>
<td>March 2019</td>
</tr>
<tr>
<td>2.7B</td>
<td>1843 HTTP Example Code Update</td>
<td>March 2019</td>
</tr>
<tr>
<td>2.7B</td>
<td>1844 Replace obsoleted RFC number with new number for TCP</td>
<td>March 2019</td>
</tr>
<tr>
<td>2.7B</td>
<td>1845 Clarification on AIP types &quot;Network boot&quot; and &quot;SAN MAC Address&quot;</td>
<td>March 2019</td>
</tr>
<tr>
<td>2.7B</td>
<td>1846 EFI_LOAD_FILE2 requirement</td>
<td>March 2019</td>
</tr>
<tr>
<td>2.7B</td>
<td>1865 Adding clarification in EFI_NOT_READY for ReadKeyStrokeEx()</td>
<td>March 2019</td>
</tr>
<tr>
<td>Revision</td>
<td>Mantis Number / Description</td>
<td>Date</td>
</tr>
<tr>
<td>----------</td>
<td>--------------------------------------------------------------------------------------------</td>
<td>-----------</td>
</tr>
<tr>
<td>2.7B</td>
<td>1869 Clarify FMP buffer too small behavior</td>
<td>March 2019</td>
</tr>
<tr>
<td>2.7B</td>
<td>1874 Add RFC3021 to reference in uefi.org</td>
<td>March 2019</td>
</tr>
<tr>
<td>2.7B</td>
<td>1875 Clarify platform specific elements in chapter 2.6.2</td>
<td>March 2019</td>
</tr>
<tr>
<td>2.7B</td>
<td>1878 Errata - Make DHCP server optional for HTTP boot</td>
<td>March 2019</td>
</tr>
<tr>
<td>2.7B</td>
<td>1880 Arm binding EL2 register state clarification</td>
<td>March 2019</td>
</tr>
<tr>
<td>2.7B</td>
<td>1890 EfiMemoryMappedIO Usage Clarification</td>
<td>March 2019</td>
</tr>
<tr>
<td>2.7B</td>
<td>1897 Clarification on mapping of UEFI memory attributes to ARM memory types and paging attributes</td>
<td>March 2019</td>
</tr>
<tr>
<td>2.7B</td>
<td>1899 Errata: Clarify EFI_INVALID_PARAMETER for FMP-&gt;GetImageInfo()</td>
<td>March 2019</td>
</tr>
<tr>
<td>2.7B</td>
<td>1901 GPT Protective MBR description</td>
<td>March 2019</td>
</tr>
<tr>
<td>2.7B</td>
<td>1902 CapsuleImageSize Clarification</td>
<td>March 2019</td>
</tr>
<tr>
<td>2.7B</td>
<td>1903 Root Directory File Name</td>
<td>March 2019</td>
</tr>
<tr>
<td>2.7B</td>
<td>1906 ACPI Table Pointer Installation</td>
<td>March 2019</td>
</tr>
<tr>
<td>2.7B</td>
<td>1908 Update of uncommitted data in the FOROM_OPEN callback</td>
<td>March 2019</td>
</tr>
<tr>
<td>2.7B</td>
<td>1923 Syntax error in EFI iSCSI Initiator Name Protocol</td>
<td>March 2019</td>
</tr>
<tr>
<td>2.7B</td>
<td>1957 Request to add status code EFI_DEVICE_ERROR for ExtractConfig</td>
<td>March 2019</td>
</tr>
<tr>
<td>2.7B</td>
<td>1964 Print disclaimer for all future UEFI specs</td>
<td>March 2019</td>
</tr>
<tr>
<td>2.7B</td>
<td>1987 incorrect VLAN_CONFIG_SET function definition</td>
<td>March 2019</td>
</tr>
<tr>
<td>2.7A</td>
<td>1830 Label Protocol - EFI_NVDIMM_LABEL_FLAGS_LOCAL definition needs to be updated</td>
<td>August 2017</td>
</tr>
<tr>
<td>2.7A</td>
<td>1829 Label Protocol Section - Missing define for EFI_NVDIMM_LABEL_FLAGS_UPDATING</td>
<td>August 2017</td>
</tr>
<tr>
<td>2.7A</td>
<td>1823 Modifications to the examples of the PCI Option ROM image combinations</td>
<td>August 2017</td>
</tr>
<tr>
<td>2.7A</td>
<td>1822 UEFI 2.7 Organization chapter duplicated</td>
<td>August 2017</td>
</tr>
<tr>
<td>2.7A</td>
<td>1821 Modify the requirement to enable PCI Bus Mastering</td>
<td>August 2017</td>
</tr>
<tr>
<td>2.7A</td>
<td>1817 NVDIMM Label Protocol - SetCookie SerialNumber needs to be UINT32 NOT UINT64</td>
<td>August 2017</td>
</tr>
<tr>
<td>2.7A</td>
<td>1816 Clarification of Using HttpConfigData in HTTP protocol</td>
<td>August 2017</td>
</tr>
<tr>
<td>2.7A</td>
<td>1815 OpenProtocol() / EFI_ALREADY_STARTED should output existent Interface</td>
<td>August 2017</td>
</tr>
<tr>
<td>2.7A</td>
<td>1808 Clarification of using option 43 in PXE v2.1</td>
<td>August 2017</td>
</tr>
<tr>
<td>2.7</td>
<td>1779 Adjusting UEFI version to UEFI 2.7</td>
<td>April 2017</td>
</tr>
<tr>
<td>2.7</td>
<td>1771BluetoothLE minor fix</td>
<td>April 2017</td>
</tr>
<tr>
<td>2.7</td>
<td>1762 UEFI UFS DEVICECONFIG Protocol</td>
<td>April 2017</td>
</tr>
<tr>
<td>2.7</td>
<td>1751 Update DNS Device Path</td>
<td>April 2017</td>
</tr>
<tr>
<td>2.7</td>
<td>1750 Add new data type to EFI Supplicant Protocol</td>
<td>April 2017</td>
</tr>
<tr>
<td>2.7</td>
<td>1745 NVDIMM Label Protocol</td>
<td>April 2017</td>
</tr>
<tr>
<td>2.7</td>
<td>1744 NVDIMMBlock Translation Table (BTT) Protocol {NewChapter}</td>
<td>April 2017</td>
</tr>
<tr>
<td>2.7</td>
<td>1730 HII Popup Protocol</td>
<td>April 2017</td>
</tr>
<tr>
<td>2.7</td>
<td>1726 Host and I/O defense</td>
<td>April 2017</td>
</tr>
<tr>
<td>2.7</td>
<td>1720 Have Partition driver publish addition information for MBR/GPT partition types.</td>
<td>April 2017</td>
</tr>
<tr>
<td>Revision</td>
<td>Mantis Number / Description</td>
<td>Date</td>
</tr>
<tr>
<td>----------</td>
<td>---------------------------------------------------------------------------------------------</td>
<td>---------</td>
</tr>
<tr>
<td>2.7</td>
<td>1719 Add EFI HTTP Boot Callback Protocol</td>
<td>April 2017</td>
</tr>
<tr>
<td>2.7</td>
<td>1718 Allow SetData to clear configuration in Ip4Config2/Ip6Config Protocol</td>
<td>April 2017</td>
</tr>
<tr>
<td>2.7</td>
<td>1716 Add BluetoothLE ECR</td>
<td>April 2017</td>
</tr>
<tr>
<td>2.7</td>
<td>1711 Firmware Error Record Update</td>
<td>April 2017</td>
</tr>
<tr>
<td>2.7</td>
<td>1707 Clarification of Private Authenticated Variables</td>
<td>April 2017</td>
</tr>
<tr>
<td>2.7</td>
<td>1701 Add wildcard support to RegisterKeyNotify</td>
<td>April 2017</td>
</tr>
<tr>
<td>2.7</td>
<td>1690 Reset Notification Protocol Update</td>
<td>April 2017</td>
</tr>
<tr>
<td>2.7</td>
<td>1689 Secure Boot with Externally Managed Configuration</td>
<td>April 2017</td>
</tr>
<tr>
<td>2.7</td>
<td>1685 Key Management Services (KMS) Protocol Enhancement</td>
<td>April 2017</td>
</tr>
<tr>
<td>2.7</td>
<td>1672 UEFI Variable Enhancements</td>
<td>April 2017</td>
</tr>
<tr>
<td>2.7</td>
<td>1654 New AIP Information block for wireless NIC</td>
<td>April 2017</td>
</tr>
<tr>
<td>2.7</td>
<td>1652 Add DNS device path node</td>
<td>April 2017</td>
</tr>
<tr>
<td>2.7</td>
<td>1647 UEFI binding for RISC-V</td>
<td>April 2017</td>
</tr>
<tr>
<td>2.7</td>
<td>1641 Simplify SecureBoot Revocation and Usage of VerifySignature</td>
<td>April 2017</td>
</tr>
<tr>
<td>2.7</td>
<td>1627 Support ASCII RegEx Patterns in EFI_REGULAR_EXPRESSION_PROTOCOL</td>
<td>April 2017</td>
</tr>
<tr>
<td>2.7</td>
<td>1627 EFI regular expression syntax type definitions</td>
<td>April 2017</td>
</tr>
<tr>
<td>2.7</td>
<td>1623 New EFI_HTTP_STATUS_CODE enum for 308 Permanent Redirect</td>
<td>April 2017</td>
</tr>
<tr>
<td>2.7</td>
<td>1623 New EFI_HTTP_STATUS_CODE enum for 308 Permanent Redirect</td>
<td>April 2017</td>
</tr>
<tr>
<td>2.6B</td>
<td>1772 Clarify EFI_NOT READY in Media State of AIP</td>
<td>April 2017</td>
</tr>
<tr>
<td>2.6B</td>
<td>1767 Incorrect structure definition for EFI_IFR_RESET_BUTTON_OP</td>
<td>April 2017</td>
</tr>
<tr>
<td>2.6B</td>
<td>1742 Clarify PK enrolling in user mode</td>
<td>April 2017</td>
</tr>
<tr>
<td>2.6B</td>
<td>1741 The memory map returnedByBS-&gt;GetMemoryMap() mayContain impossible values.</td>
<td>April 2017</td>
</tr>
<tr>
<td>2.6B</td>
<td>1739 typos -Broken references link.</td>
<td>April 2017</td>
</tr>
<tr>
<td>2.6B</td>
<td>1729Cleanup of ACPI 2.0 references in UEFI spec</td>
<td>April 2017</td>
</tr>
<tr>
<td>2.6B</td>
<td>1708 Typos in Image Decode and Image Ex Protocols</td>
<td>April 2017</td>
</tr>
<tr>
<td>2.6B</td>
<td>1700 Align ACPI descriptor definitions in PCI I/O and PCI RootBridge I/O</td>
<td>April 2017</td>
</tr>
<tr>
<td>2.6B</td>
<td>1698 Update to Mantis 1613 - GetNextVariable</td>
<td>April 2017</td>
</tr>
<tr>
<td>2.6B</td>
<td>1691 Remove/Deprecate SMM Communication ACPI Table</td>
<td>April 2017</td>
</tr>
<tr>
<td>2.6B</td>
<td>1682 HII Protocol StatusCodes</td>
<td>April 2017</td>
</tr>
<tr>
<td>2.6B</td>
<td>1678 Simplify the ACPI Table GUID declarations</td>
<td>April 2017</td>
</tr>
<tr>
<td>2.6B</td>
<td>1675 section 30.5.1 typo</td>
<td>April 2017</td>
</tr>
<tr>
<td>2.6B</td>
<td>1668 Duplicate GUID issue - mustChange the Image Decoder Protocol GUID</td>
<td>April 2017</td>
</tr>
<tr>
<td>2.6B</td>
<td>1655 HTTP errata inConfigure()</td>
<td>April 2017</td>
</tr>
<tr>
<td>2.6B</td>
<td>1653 Incorrect errorCode value in MTFTP6</td>
<td>April 2017</td>
</tr>
<tr>
<td>2.6B</td>
<td>1634 Update to the EFI_SIMPLE_TEXT_INPUT_PROTOCOL TPL restriction</td>
<td>April 2017</td>
</tr>
<tr>
<td>2.6B</td>
<td>1629 Errata in GetVariable description</td>
<td>April 2017</td>
</tr>
<tr>
<td>2.6B</td>
<td>1625 Clarification of HTTPBoot wire protocol “HTTPClient” VendorClass Option</td>
<td>April 2017</td>
</tr>
<tr>
<td>2.6B</td>
<td>1624 Fix spelling typo in EFI_HTTP_STATUS_CODE</td>
<td>April 2017</td>
</tr>
<tr>
<td>Revision</td>
<td>Mantis Number / Description</td>
<td>Date</td>
</tr>
<tr>
<td>----------</td>
<td>----------------------------</td>
<td>------------</td>
</tr>
<tr>
<td>2.6B</td>
<td>1613 GetNextVariableName Errata</td>
<td>April 2017</td>
</tr>
<tr>
<td>2.6B</td>
<td>1612 ResetSystem Errata</td>
<td>April 2017</td>
</tr>
<tr>
<td>2.6B</td>
<td>1609 UEFI Errata - Address Security problems in the Pkcs7Verify Protocol</td>
<td>April 2017</td>
</tr>
<tr>
<td>2.6B</td>
<td>1608 Enhance EFI_IFR_NUMERIC (Step)</td>
<td>April 2017</td>
</tr>
<tr>
<td>2.6B</td>
<td>1586 Errors in appendix N for ARM ProcessorContext Information</td>
<td>April 2017</td>
</tr>
<tr>
<td>2.6B</td>
<td>1584 WiFi errata</td>
<td>April 2017</td>
</tr>
<tr>
<td>2.6B</td>
<td>1580 Correct some typos</td>
<td>April 2017</td>
</tr>
<tr>
<td>2.6B</td>
<td>1559 Clarify return value for NULL pointer in LocateProtocol() API</td>
<td>April 2017</td>
</tr>
<tr>
<td>2.6B</td>
<td>1557 secureBoot and auth variable errata</td>
<td>April 2017</td>
</tr>
<tr>
<td>2.6B</td>
<td>1556 HTTPv6Boot DHCP Options Errata</td>
<td>April 2017</td>
</tr>
<tr>
<td>2.6B</td>
<td>1555 USB Function port protocol errata</td>
<td>April 2017</td>
</tr>
<tr>
<td>2.6B</td>
<td>1554 fix to ecr 1539</td>
<td>April 2017</td>
</tr>
<tr>
<td>2.6B</td>
<td>1553 os recoveryBoot option errata</td>
<td>April 2017</td>
</tr>
<tr>
<td>2.6B</td>
<td>1551 EFIBluetoothConfiguration Protocol Errata</td>
<td>April 2017</td>
</tr>
<tr>
<td>2.6B</td>
<td>1550 Replace FTP4 dataCallback pointer-to-function-pointer with regular function pointer</td>
<td>April 2017</td>
</tr>
<tr>
<td>2.6A</td>
<td>SameContent as version 2.6,But with the Adobe “accessibility” feature activated so text-to-speech will work.</td>
<td>December 2016</td>
</tr>
<tr>
<td>2.6</td>
<td>1548ClarifyBoot procedure when file name is absent2.</td>
<td>January, 2016</td>
</tr>
<tr>
<td>2.6</td>
<td>1547 Clarify requirements for setting the PK variable.</td>
<td>January, 2016</td>
</tr>
<tr>
<td>2.6</td>
<td>1544 DNS lookup API spelling</td>
<td>January, 2016</td>
</tr>
<tr>
<td>2.6</td>
<td>1543 ip4/6Config policy errata/2.6 update</td>
<td>January, 2016</td>
</tr>
<tr>
<td>2.6</td>
<td>1542 UEFI 2.6 supplicant errata</td>
<td>January, 2016</td>
</tr>
<tr>
<td>2.6</td>
<td>1539 New EFI_HTTP_ERROR StatusCode</td>
<td>December, 2015</td>
</tr>
<tr>
<td>2.6</td>
<td>1538 UEFI TLS errata</td>
<td>December, 2015</td>
</tr>
<tr>
<td>2.6</td>
<td>1536 UEFI 2.6 Errata : IMAGE EX Protocol and EFI HII Image Decoder protocol Errata</td>
<td>December, 2015</td>
</tr>
<tr>
<td>2.6</td>
<td>1534 EditorialComments against 2.6 Final Draft</td>
<td>December, 2015</td>
</tr>
<tr>
<td>2.6</td>
<td>1533Bugs in the HTTP usage example</td>
<td>December, 2015</td>
</tr>
<tr>
<td>2.6</td>
<td>1523Comments against 2.6 Draft</td>
<td>December, 2015</td>
</tr>
<tr>
<td>2.6</td>
<td>1522 AArch64Bindings AlignmentBit errata</td>
<td>December, 2015</td>
</tr>
<tr>
<td>2.6</td>
<td>1521Comment against UEFI.next draft - M1479</td>
<td>December, 2015</td>
</tr>
<tr>
<td>2.6</td>
<td>1519 Version for the next UEFI spec is...</td>
<td>December, 2015</td>
</tr>
<tr>
<td>2.6</td>
<td>1518Comments against 2.6 Draft</td>
<td>December, 2015</td>
</tr>
<tr>
<td>2.6</td>
<td>1516 EditorialComments against 2.6 Draft</td>
<td>December, 2015</td>
</tr>
<tr>
<td>2.6</td>
<td>1509 EFI_PLATFORM_TO_DRIVER_CONFIGURATION_PROTOCOL Response to unsupported ParameterTypeGuid</td>
<td>December, 2015</td>
</tr>
<tr>
<td>2.6</td>
<td>1508 Lack of flexibility and realism in exception levelChoice whenCalling runtime services</td>
<td>December, 2015</td>
</tr>
<tr>
<td>2.6</td>
<td>1507 Insufficient qualification of page attributes for AArch64</td>
<td>December, 2015</td>
</tr>
<tr>
<td>Revision</td>
<td>Mantis Number / Description</td>
<td>Date</td>
</tr>
<tr>
<td>----------</td>
<td>----------------------------</td>
<td>---------------</td>
</tr>
<tr>
<td>2.6</td>
<td>1502 PCI IO Define how to use the Address Translation Offset for systems that are not mapped 1:1</td>
<td>November, 2015</td>
</tr>
<tr>
<td>2.6</td>
<td>1501 Define the usage of the &quot;Address Space Granularity&quot; field is defined in the PCI Root IO</td>
<td>November, 2015</td>
</tr>
<tr>
<td>2.6</td>
<td>1496Bad table reference in 13.2 EFI_PCI_ROOT_BRIDGE_IO_PROTOCOL.Configuration()</td>
<td>November, 2015</td>
</tr>
<tr>
<td>2.6</td>
<td>1494 Errata against UEFI 2.5 Properties Table</td>
<td>November, 2015</td>
</tr>
<tr>
<td>2.6</td>
<td>1493 Updates to the SD_MMC_PASS_THRU interface</td>
<td>November, 2015</td>
</tr>
<tr>
<td>2.6</td>
<td>1492 wireless macConnection protocol li errata</td>
<td>November, 2015</td>
</tr>
<tr>
<td>2.6</td>
<td>1491 supplicant errata</td>
<td>November, 2015</td>
</tr>
<tr>
<td>2.6</td>
<td>1480 Refine Progress description in EFI_KEYWOD_HANDLER_PROTOCOL</td>
<td>November, 2015</td>
</tr>
<tr>
<td>2.6</td>
<td>1479 UEFI Properties TableClarification</td>
<td>November, 2015</td>
</tr>
<tr>
<td>2.6</td>
<td>1471 SD/eMMC PassThru Protocol update (follow up to mantis 1376)</td>
<td>November, 2015</td>
</tr>
<tr>
<td>2.6</td>
<td>1467 New API - EFI_WIRELESS_MAC_CONNECTION_II_PROTOCOL</td>
<td>November, 2015</td>
</tr>
<tr>
<td>2.6</td>
<td>1466 UEFI Ram disk protocol</td>
<td>November, 2015</td>
</tr>
<tr>
<td>2.6</td>
<td>1452 Minor edits to 0001409</td>
<td>November, 2015</td>
</tr>
<tr>
<td>2.6</td>
<td>1414 Generalisation ofCommunication method in Appendix O</td>
<td>November, 2015</td>
</tr>
<tr>
<td>2.6</td>
<td>1409 EFI HII ImageEX protocol and EFI HII Image Decoder protocols</td>
<td>November, 2015</td>
</tr>
<tr>
<td>2.6</td>
<td>1408 EFI HII Font EX protocol and EFI HII Font Glyph Generator protocols</td>
<td>November, 2015</td>
</tr>
<tr>
<td>2.6</td>
<td>1402 Add EFI_BROWSER_ACTION_SUBMITTED</td>
<td>November, 2015</td>
</tr>
<tr>
<td>2.6</td>
<td>1383 Adding an EraseBlocks() function to a new protocol</td>
<td>November, 2015</td>
</tr>
<tr>
<td>2.6</td>
<td>1376 SD/eMMC PassThru Protocol</td>
<td>November, 2015</td>
</tr>
<tr>
<td>2.6</td>
<td>1357 ARMCPER extensions</td>
<td>November, 2015</td>
</tr>
<tr>
<td>2.5A</td>
<td>1481 new networkConfig2 protocol data structure has a magic number</td>
<td>October 2015</td>
</tr>
<tr>
<td>2.5A</td>
<td>1477 AllowCloseEvent toBeCalled within the Notification Function</td>
<td>October 2015</td>
</tr>
<tr>
<td>2.5A</td>
<td>1476 Update to Indicate thatCloseEvent UnregistersCorresponding Protocol Notification Registrations</td>
<td>October 2015</td>
</tr>
<tr>
<td>2.5A</td>
<td>1472 ATA Pass Thru Errata</td>
<td>October 2015</td>
</tr>
<tr>
<td>2.5A</td>
<td>1469 UNDI Errata - add more statistics</td>
<td>October 2015</td>
</tr>
<tr>
<td>2.5A</td>
<td>1468 Errata on UEFI Suppliant protocol</td>
<td>October 2015</td>
</tr>
<tr>
<td>2.5A</td>
<td>1451 Memory MapConsistency</td>
<td>October 2015</td>
</tr>
<tr>
<td>2.5A</td>
<td>1441 UEFI2.5A – UNDI ProtocolClarification</td>
<td>October 2015</td>
</tr>
<tr>
<td>2.5A</td>
<td>1426 UEFI 2.5 typo</td>
<td>October 2015</td>
</tr>
<tr>
<td>2.5A</td>
<td>1424 Incorrect link in Section 22.1 FMP GetImageInfo()</td>
<td>October 2015</td>
</tr>
<tr>
<td>2.5A</td>
<td>1421 Misc HTTP API typos</td>
<td>October 2015</td>
</tr>
<tr>
<td>2.5A</td>
<td>1420 GetNextHighMonotonicCountClarification</td>
<td>October 2015</td>
</tr>
<tr>
<td>2.5A</td>
<td>1419 Suppliant protocol using same GUID as TLS protocol</td>
<td>October 2015</td>
</tr>
<tr>
<td>2.5A</td>
<td>1418 Inconsistent issues in DNS</td>
<td>October 2015</td>
</tr>
<tr>
<td>2.5A</td>
<td>1417 Add HttpMethodMax to EFI_HTTP_METHOD enum</td>
<td>October 2015</td>
</tr>
<tr>
<td>2.5A</td>
<td>1410 Clarifications in appendix O</td>
<td>October 2015</td>
</tr>
<tr>
<td>Revision</td>
<td>Mantis Number / Description</td>
<td>Date</td>
</tr>
<tr>
<td>----------</td>
<td>--------------------------------------------------------------------------------------------</td>
<td>------------</td>
</tr>
<tr>
<td>2.5A</td>
<td>1407 Networking errata - EFI_HTTP_STATUS typos</td>
<td>October 2015</td>
</tr>
<tr>
<td>2.5A</td>
<td>1405 Errata in table 271 in Appendix O</td>
<td>October 2015</td>
</tr>
<tr>
<td>2.5A</td>
<td>1399 Clarification for EFI_BROWSER_ACTION_REQUEST_RECONNECT</td>
<td>October 2015</td>
</tr>
<tr>
<td>2.5A</td>
<td>1398 Errata update to the runtime GetVariable operation documentation</td>
<td>October 2015</td>
</tr>
<tr>
<td>2.5A</td>
<td>1388 Missed memory type fixes</td>
<td>October 2015</td>
</tr>
<tr>
<td>2.5A</td>
<td>1381 Remove informativeContent in 12.6.1</td>
<td>October 2015</td>
</tr>
<tr>
<td>2.5A</td>
<td>1365 7.4 Virtual Memory Services lists Section 2.3.2 through Section 2.3.4. incorrectly</td>
<td>October 2015</td>
</tr>
<tr>
<td>2.5A</td>
<td>1363 Short form URI device path</td>
<td>October 2015</td>
</tr>
<tr>
<td>2.5A</td>
<td>1209 UEFI networking API Chapter 2.6 requirements errors</td>
<td>October 2015</td>
</tr>
<tr>
<td>2.5A</td>
<td></td>
<td>October 2015</td>
</tr>
<tr>
<td>2.5</td>
<td>1364 Extend supplicant data type for EAP</td>
<td>April, 2015</td>
</tr>
<tr>
<td>2.5</td>
<td>1362 HTTPBoot typos/bugs</td>
<td>April, 2015</td>
</tr>
<tr>
<td>2.5</td>
<td>1360 Vendor Range for UEFI memory Types</td>
<td>April, 2015</td>
</tr>
<tr>
<td>2.5</td>
<td>1358 v2.5 amendment and v2.4 errata (missed implementation of Mantis 1089)</td>
<td>April, 2015</td>
</tr>
<tr>
<td>2.5</td>
<td>1353 SATA Device Path Node Errata</td>
<td>April, 2015</td>
</tr>
<tr>
<td>2.5</td>
<td>1352 Errata for 1263 and 1227</td>
<td>April, 2015</td>
</tr>
<tr>
<td>2.5</td>
<td>1350 Keyword Strings Errata</td>
<td>April, 2015</td>
</tr>
<tr>
<td>2.5</td>
<td>1348 ERRATA - Section 10.12 EFI_ADAPTER_INFORMATION_PROTOCOL Custom Types</td>
<td>April, 2015</td>
</tr>
<tr>
<td>2.5</td>
<td>1347 Boot Manager Policy Errata</td>
<td>April, 2015</td>
</tr>
<tr>
<td>2.5</td>
<td>1346 Mantis 1288 Errata</td>
<td>April, 2015</td>
</tr>
<tr>
<td>2.5</td>
<td>1345 EFI_USB2_HC_PROTOCOL Errata</td>
<td>April, 2015</td>
</tr>
<tr>
<td>2.5</td>
<td>1342 DNS6 - friendly amendment for review By USWG</td>
<td>April, 2015</td>
</tr>
<tr>
<td>2.5</td>
<td>1341 DNS4 - friendly amendment to Be reviewed By USWG</td>
<td>April, 2015</td>
</tr>
<tr>
<td>2.5</td>
<td>1339 Errata in section 7.2.3.2 Hardware Error Record Variables</td>
<td>April, 2015</td>
</tr>
<tr>
<td>2.5</td>
<td>1309 Disallow EFI_VARIABLE_AUTHENTICATION from SecureBoot Policy Variables</td>
<td>April, 2015</td>
</tr>
<tr>
<td>2.5</td>
<td>1308 Fix typo’s found in the final/published UEFI 2.4 ErrataB spec</td>
<td>February, 2015</td>
</tr>
<tr>
<td>2.5</td>
<td>1304 Add IMAGE_UPDATABLE_VALID_WITH_VENDOR_CODE to FMPCheck image</td>
<td>February, 2015</td>
</tr>
<tr>
<td>2.5</td>
<td>1303 Update the UEFI version to reflect new revision</td>
<td>February, 2015</td>
</tr>
<tr>
<td>2.5</td>
<td>1288 The Macro definition Conflict in EFI_SIMPLE_TEXT_OUTPUT_PROTOCOL_SetAttribute() in UEFI 2.4B</td>
<td>February, 2015</td>
</tr>
<tr>
<td>2.5</td>
<td>1287 Errata: EFI Driver Supported EFI Version not matching the spec revision</td>
<td>February, 2015</td>
</tr>
<tr>
<td>2.5</td>
<td>1269 Configuration Routing Protocol and Configuration String Updates</td>
<td>February, 2015</td>
</tr>
<tr>
<td>2.5</td>
<td>1268 RAM Disk UEFI Device Path Node</td>
<td>February, 2015</td>
</tr>
<tr>
<td>2.5</td>
<td>1266 UEFI.Next Feature - IP_CONFIG2 Protocol</td>
<td>February, 2015</td>
</tr>
<tr>
<td>2.5</td>
<td>1263 Customized Deployment of SecureBoot</td>
<td>February, 2015</td>
</tr>
<tr>
<td>2.5</td>
<td>1257 Correct the typedef definitions for EFI_BOOT_SERVICES/EFI_RUNTIME_SERVICES-Reiterate</td>
<td>February, 2015</td>
</tr>
<tr>
<td>2.5</td>
<td>1255 UFS Device Path Node Length</td>
<td>February, 2015</td>
</tr>
<tr>
<td>2.5</td>
<td>1254 SD Device Path</td>
<td>February, 2015</td>
</tr>
<tr>
<td>2.5</td>
<td>1251 EFI_REGULAR_EXPRESSION_PROTOCOL and EFI_IFR_MATCH2 HII op-code</td>
<td>February, 2015</td>
</tr>
<tr>
<td>Revision</td>
<td>Mantis Number / Description</td>
<td>Date</td>
</tr>
<tr>
<td>----------</td>
<td>------------------------------------------------------------------------------------------------</td>
<td>-------------</td>
</tr>
<tr>
<td>2.5</td>
<td>1244 sections of the spec mis-arranged</td>
<td>February, 2015</td>
</tr>
<tr>
<td>2.5</td>
<td>1234 UEFI.Next feature - SmartCard edge protocol</td>
<td>February, 2015</td>
</tr>
<tr>
<td>2.5</td>
<td>1227 UEFI.Next feature - Platform recovery</td>
<td>February, 2015</td>
</tr>
<tr>
<td>2.5</td>
<td>1224 UEFI.Next - Adding support for No executable data areas</td>
<td>February, 2015</td>
</tr>
<tr>
<td>2.5</td>
<td>1223 UEFI.Next networking features - Chapter 2.6 requirements</td>
<td>February, 2015</td>
</tr>
<tr>
<td>2.5</td>
<td>1222 UEFI.Next feature - BMC/Service Processor Device Path</td>
<td>February, 2015</td>
</tr>
<tr>
<td>2.5</td>
<td>1221 UEFI.Next feature - REST Protocol</td>
<td>February, 2015</td>
</tr>
<tr>
<td>2.5</td>
<td>1220 UEFI.Next feature - Bluetooth</td>
<td>February, 2015</td>
</tr>
<tr>
<td>2.5</td>
<td>1219 UEFI.Next Feature - UEFI TLS API</td>
<td>February, 2015</td>
</tr>
<tr>
<td>2.5</td>
<td>1218 UEFI.Next feature - EAP2 Protocol</td>
<td>February, 2015</td>
</tr>
<tr>
<td>2.5</td>
<td>1217 UEFI.Next feature - WiFi support</td>
<td>February, 2015</td>
</tr>
<tr>
<td>2.5</td>
<td>1216 UEFI.next feature - DNS version 6</td>
<td>February, 2015</td>
</tr>
<tr>
<td>2.5</td>
<td>1215 UEFI.Next feature - DNS version 4</td>
<td>February, 2015</td>
</tr>
<tr>
<td>2.5</td>
<td>1214 UEFI.Next feature - HTTPBoot</td>
<td>February, 2015</td>
</tr>
<tr>
<td>2.5</td>
<td>1213 UEFI.Next feature - HTTP helper API</td>
<td>February, 2015</td>
</tr>
<tr>
<td>2.5</td>
<td>1212 UEFI.Next feature - HTTP API</td>
<td>February, 2015</td>
</tr>
<tr>
<td>2.5</td>
<td>1204 new UEFI USB Function I/O Protocol addition to the UEFI spec</td>
<td>February, 2015</td>
</tr>
<tr>
<td>2.5</td>
<td>1201 Exposing Memory Redundancy to OSPM</td>
<td>February, 2015</td>
</tr>
<tr>
<td>2.5</td>
<td>1199 Add NVM Express Pass Thru Protocol</td>
<td>February, 2015</td>
</tr>
<tr>
<td>2.5</td>
<td>1191 Add new SMBIOS3_TABLE_GUID in EFI_CONFIGURATION_TABLE</td>
<td>February, 2015</td>
</tr>
<tr>
<td>2.5</td>
<td>1186 AArch64 Binding Clarifications and errata</td>
<td>February, 2015</td>
</tr>
<tr>
<td>2.5</td>
<td>1183 New Protocol with 2 Function for PKCS7 Signature Verification Services</td>
<td>February, 2015</td>
</tr>
<tr>
<td>2.5</td>
<td>1174 errata - Error in EFI_IFR_PASSWORD logic flowchart</td>
<td>February, 2015</td>
</tr>
<tr>
<td>2.5</td>
<td>1167 Persistent Memory Type support</td>
<td>February, 2015</td>
</tr>
<tr>
<td>2.5</td>
<td>1166 hash 2 protocol errata</td>
<td>February, 2015</td>
</tr>
<tr>
<td>2.5</td>
<td>1163 InlineCryptographic Interface Protocol proposal</td>
<td>February, 2015</td>
</tr>
<tr>
<td>2.5</td>
<td>1159 Proposal for System Prep Applications</td>
<td>February, 2015</td>
</tr>
<tr>
<td>2.5</td>
<td>1158 errata - Boot manager Clarification</td>
<td>February, 2015</td>
</tr>
<tr>
<td>2.5</td>
<td>1147--REDACT</td>
<td>February, 2015</td>
</tr>
<tr>
<td>2.5</td>
<td>1121 IPV6 support from UNDI</td>
<td>February, 2015</td>
</tr>
<tr>
<td>2.5</td>
<td>1109 SmartCard Reader</td>
<td>February, 2015</td>
</tr>
<tr>
<td>2.5</td>
<td>1103 Longer term New CPER Memory Section</td>
<td>February, 2015</td>
</tr>
<tr>
<td>2.5</td>
<td>1091 Clarification of handle to host FMP</td>
<td>February, 2015</td>
</tr>
<tr>
<td>2.5</td>
<td>1090 ESRT: EFI System Resource Table and Component firmware updates</td>
<td>February, 2015</td>
</tr>
<tr>
<td>2.5</td>
<td>1071 New EFI_HASH2_PROTOCOL</td>
<td>February, 2015</td>
</tr>
<tr>
<td>2.4C</td>
<td>1308 Fix typo's found in the final/published UEFI 2.4 ErrataB spec</td>
<td>January 2015</td>
</tr>
<tr>
<td>2.4C</td>
<td>1287 Errata: EFI Driver Supported EFI Version not matching the spec revision</td>
<td>January 2015</td>
</tr>
<tr>
<td>2.4C</td>
<td>1257 Correct the typedef definitions for EFI_BOOT_SERVICES/EFI_RUNTIME_SERVICES</td>
<td>January 2015</td>
</tr>
<tr>
<td>2.4C</td>
<td>1244 sections of the spec misarranged</td>
<td>January 2015</td>
</tr>
<tr>
<td>Revision</td>
<td>Mantis Number / Description</td>
<td>Date</td>
</tr>
<tr>
<td>----------</td>
<td>--------------------------------------------------------------------------------------------</td>
<td>-----------------</td>
</tr>
<tr>
<td>2.4C</td>
<td>1211 EFI_LOAD_OPTION Definition</td>
<td>January 2015</td>
</tr>
<tr>
<td>2.4C</td>
<td>1209 Errata - UEFI networking API Chapter 2.6 requirements</td>
<td>January 2015</td>
</tr>
<tr>
<td>2.4C</td>
<td>1205 Errata for Hii Set item</td>
<td>January 2015</td>
</tr>
<tr>
<td>2.4C</td>
<td>1200 Universal Flash Storage (UFS) Device Path</td>
<td>January 2015</td>
</tr>
<tr>
<td>2.4C</td>
<td>1198 EFI_ATA_PASS_THRU_PROTOCOL Clarification</td>
<td>January 2015</td>
</tr>
<tr>
<td>2.4C</td>
<td>1194 Add EFI_IFR_FLAG_RECONNECT_REQUIRED</td>
<td>January 2015</td>
</tr>
<tr>
<td>2.4C</td>
<td>1192 Cleanup GUID formatting issues</td>
<td>January 2015</td>
</tr>
<tr>
<td>2.4C</td>
<td>1186 AArch64BindingClarifications and errata</td>
<td>January 2015</td>
</tr>
<tr>
<td>2.4C</td>
<td>1185 errata - tcp api</td>
<td>January 2015</td>
</tr>
<tr>
<td>2.4C</td>
<td>1184 errata - snp mode Clarification</td>
<td>January 2015</td>
</tr>
<tr>
<td>2.4C</td>
<td>1182 Errata - UEFI URI Device path issue</td>
<td>January 2015</td>
</tr>
<tr>
<td>2.4C</td>
<td>1174 errata - Error in EFI_IFR_PASSWORD logic flowchart</td>
<td>January 2015</td>
</tr>
<tr>
<td>2.4C</td>
<td>1173 EFI_IFR_NUMERIC Errata</td>
<td>July 11, 2014</td>
</tr>
<tr>
<td>2.4C</td>
<td>1172 EfACPIMemoryNVS definition missing S4</td>
<td>July 11, 2014</td>
</tr>
<tr>
<td>2.4C</td>
<td>1170 Errata pxeBc api Clarification</td>
<td>July 11, 2014</td>
</tr>
<tr>
<td>2.4C</td>
<td>1169 Errata - volatile networking variable Cleanup</td>
<td>July 11, 2014</td>
</tr>
<tr>
<td>2.4C</td>
<td>1168 MTFTP Errata</td>
<td>July 11, 2014</td>
</tr>
<tr>
<td>2.4C</td>
<td>1165 Option rom layout errata</td>
<td>July 11, 2014</td>
</tr>
<tr>
<td>2.4C</td>
<td>1162 Typo in ReinstallProtocolInterface() EFI 1.10 Extension section</td>
<td>July 11, 2014</td>
</tr>
<tr>
<td>2.4C</td>
<td>1150 Missing LineBreakCharacter (Hii Errata)</td>
<td>July 11, 2014</td>
</tr>
<tr>
<td>2.4C</td>
<td>1147 EFI_USB2_HC_PROTOCOL.AsyncInterruptTransfer() Errata</td>
<td>July 11, 2014</td>
</tr>
<tr>
<td>2.4C</td>
<td>1141 UEFI errata - ia32/x64 vector register management</td>
<td>July 11, 2014</td>
</tr>
<tr>
<td>2.4C</td>
<td>1140 UEFI Errata - image execution info table</td>
<td>July 11, 2014</td>
</tr>
<tr>
<td>2.4C</td>
<td>1139 UEFI Errata on the storage securityCommand protocol</td>
<td>July 11, 2014</td>
</tr>
<tr>
<td>2.4C</td>
<td>1066 Errata--reference to missing table (90) removed</td>
<td>July 11, 2014</td>
</tr>
<tr>
<td>2.4C</td>
<td>1043 Ability to refresh the entire form [newContent]</td>
<td>July 11, 2014</td>
</tr>
<tr>
<td>2.4C</td>
<td>1042 AddBrowser Action Request &quot;reconnect&quot;</td>
<td>July 11, 2014</td>
</tr>
<tr>
<td>2.4B</td>
<td>1146 Typos and Broken links</td>
<td>April 17, 2014</td>
</tr>
<tr>
<td>2.4B</td>
<td>1137 Typographic errors in the 2.4 Errata B draft</td>
<td>April 16, 2014</td>
</tr>
<tr>
<td>2.4B</td>
<td>1128 URI device path node redux--supersedes (defunct) 1119</td>
<td>April 4, 2014</td>
</tr>
<tr>
<td>2.4B</td>
<td>1127 USB Errata - unnecessary restriction on UEFI interrupt transfer types</td>
<td>March 27, 2014</td>
</tr>
<tr>
<td>2.4B</td>
<td>1124 Adding text description for NVMe device node</td>
<td>March 27, 2014</td>
</tr>
<tr>
<td>2.4B</td>
<td>1122 Correct misleading language in the UEFI 2.4a specification about the</td>
<td>March 27, 2014</td>
</tr>
<tr>
<td></td>
<td>EFI_ADAPTER_INFORMATION_PROTOCOL.EFI_ADAPTER_INFO_GET_SUPPORTED_TYPES function</td>
<td></td>
</tr>
<tr>
<td>2.4B</td>
<td>1120 Make time stamp handling Consistent around all of the networking API’s</td>
<td>March 27, 2014</td>
</tr>
<tr>
<td>2.4B</td>
<td>1118 Network Performance Enhancements Concerning Volatile Variables</td>
<td>March 27, 2014</td>
</tr>
<tr>
<td>2.4B</td>
<td>1115 Clarification on the usage of XMM/FPU instructions from within a UEFI Runtime Service on</td>
<td>March 27, 2014</td>
</tr>
<tr>
<td></td>
<td>an x64 processor</td>
<td></td>
</tr>
<tr>
<td>2.4B</td>
<td>1111 Errors in DisconnectController() returnCode descriptions</td>
<td>March 27, 2014</td>
</tr>
<tr>
<td>Revision</td>
<td>Mantis Number / Description</td>
<td>Date</td>
</tr>
<tr>
<td>----------</td>
<td>-----------------------------</td>
<td>------------</td>
</tr>
<tr>
<td>2.4B</td>
<td>1101 Errata – ReinstallProtocolInterface</td>
<td>March 27, 2014</td>
</tr>
<tr>
<td>2.4B</td>
<td>1092 Clarification to PCI Option ROM Driver Loading Description</td>
<td>March 27, 2014</td>
</tr>
<tr>
<td>2.4B</td>
<td>1085 Error--added in missing text approved for 2.4A</td>
<td>April 17, 2014</td>
</tr>
<tr>
<td>2.4B</td>
<td>1014 HIIConfig Access Protocol Errata</td>
<td>April 3, 2014</td>
</tr>
<tr>
<td>2.4 A</td>
<td>1089 Short-termCPER Memory Section errata</td>
<td>Nov. 14, 2013</td>
</tr>
<tr>
<td>2.4 A</td>
<td>1088 Add revision #define to EFI_FILE_PROTOCOL</td>
<td>Nov. 6, 2013</td>
</tr>
<tr>
<td>2.4 A</td>
<td>1085 Issues with Interactive password</td>
<td>Nov. 14, 2013</td>
</tr>
<tr>
<td>2.4 A</td>
<td>1082 Mistake in 2.3.5.1 / 2.3.6.2 Handoff State</td>
<td>Nov. 6, 2013</td>
</tr>
<tr>
<td>2.4 A</td>
<td>1081 Update Install Table protocol to deal with duplicate tables</td>
<td>Nov. 6, 2013</td>
</tr>
<tr>
<td>2.4 A</td>
<td>1079 UEFI 2.4: Remove repetitive “the” (typo)</td>
<td>Nov. 6, 2013</td>
</tr>
<tr>
<td>2.4 A</td>
<td>1078 Adjust some text for handling EFI_BROWSE ACTION_CHANGING</td>
<td>Nov. 6, 2013</td>
</tr>
<tr>
<td>2.4 A</td>
<td>1077 Fix wording in EVT_SIGNAL_EXIT_BOOT_SERVICES</td>
<td>Nov. 6, 2013</td>
</tr>
<tr>
<td>2.4 A</td>
<td>1076 typo in UEFI v2.3.1d and v2.4</td>
<td>Nov. 6, 2013</td>
</tr>
<tr>
<td>2.4 A</td>
<td>1075 Clarifications to Table 88. Device Node Table (Device Node to TextConversion)</td>
<td>Nov. 6, 2013</td>
</tr>
<tr>
<td>2.4 A</td>
<td>1074 Add Clarifications on DMA requirements for PCI_IO</td>
<td>Nov. 6, 2013</td>
</tr>
<tr>
<td>2.4 A</td>
<td>1073 Add requirement for EFI_USB_IO_PROTOCOL</td>
<td>Nov. 6, 2013</td>
</tr>
<tr>
<td>2.4 A</td>
<td>1066 Errata - ISCSI IPV6 Root Path Clarification</td>
<td>Nov. 6, 2013</td>
</tr>
<tr>
<td>2.4 A</td>
<td>1064 AIP Errata</td>
<td>Nov. 6, 2013</td>
</tr>
<tr>
<td>2.4 A</td>
<td>1063 Correction to GPT expression for SizeofPartitionEntry</td>
<td>Nov. 6, 2013</td>
</tr>
<tr>
<td>2.4 A</td>
<td>1062 EFI_CERT_X509_GUID does not specify the Certificate encoding</td>
<td>Nov. 6, 2013</td>
</tr>
<tr>
<td>2.4 A</td>
<td>1061 UEFI 2.4 section 2.6.2 and 2.6.3 don’t use protocol hyperlinksConsistently</td>
<td>Nov. 6, 2013</td>
</tr>
<tr>
<td>2.4 A</td>
<td>1060 Slight Clarification to FMP Authentication Requirements</td>
<td>Nov. 6, 2013</td>
</tr>
<tr>
<td>2.4 A</td>
<td>1059 Clarification of a return status Code of HASH protocol</td>
<td>Nov. 6, 2013</td>
</tr>
<tr>
<td>2.4 A</td>
<td>1058 Correct mistake in the system table revision</td>
<td>Nov. 6, 2013</td>
</tr>
<tr>
<td>2.4 A</td>
<td>1056 text modification to definition of EFI_FIRMWARE_IMAGE_DESCRIPTOR_VERSION 2</td>
<td>Nov. 6, 2013</td>
</tr>
<tr>
<td>2.4 A</td>
<td>1055 Disk IO 2 errata</td>
<td>Nov. 6, 2013</td>
</tr>
<tr>
<td>2.4 A</td>
<td>1054 Deprecate 6 Hash Algorithms with inconsistent usage</td>
<td>Nov. 6, 2013</td>
</tr>
<tr>
<td>2.4 A</td>
<td>1053 Reduce Name space ofCapsule Result variable to increase performance</td>
<td>Nov. 6, 2013</td>
</tr>
<tr>
<td>2.4 A</td>
<td>1035 PCI Option ROM Errata (five figures)</td>
<td>Nov. 6, 2013</td>
</tr>
<tr>
<td>2.4</td>
<td>997 Driver Health Protocol errorCodes</td>
<td>April 25, 2013</td>
</tr>
<tr>
<td>2.4</td>
<td>993 (original ticket--supersededBy 1026)</td>
<td>April 25, 2013</td>
</tr>
<tr>
<td>2.4</td>
<td>992 Adapter Information Protocol (AIP)</td>
<td>April 25, 2013</td>
</tr>
<tr>
<td>2.4</td>
<td>991 Greater than 256 NICs support on UNDI</td>
<td>April 25, 2013</td>
</tr>
<tr>
<td>2.4</td>
<td>968 HII Forms op-code for displaying a warning message</td>
<td>April 25, 2013</td>
</tr>
<tr>
<td>2.4</td>
<td>966 Spec typos</td>
<td>April 25, 2013</td>
</tr>
<tr>
<td>2.4</td>
<td>964 Disk IO 2 Protocol to support Async IO</td>
<td>April 25, 2013</td>
</tr>
<tr>
<td>2.4</td>
<td>963 Add new device path node NVM Express devices</td>
<td>April 25, 2013</td>
</tr>
<tr>
<td>2.4</td>
<td>956 Require network drivers to return EFI_NO_MEDIA</td>
<td>April 25, 2013</td>
</tr>
<tr>
<td>2.4</td>
<td>946 Forbid Creation of non-spec variables in EFI_GLOBAL_VARIABLE namespace</td>
<td>April 25, 2013</td>
</tr>
<tr>
<td>Revision</td>
<td>Mantis Number / Description</td>
<td>Date</td>
</tr>
<tr>
<td>----------</td>
<td>------------------------------------------------------------------------------------------</td>
<td>--------------</td>
</tr>
<tr>
<td>2.4</td>
<td>920 Add a variable for indicating out of Band key modification</td>
<td>April 25, 2013</td>
</tr>
<tr>
<td>2.4</td>
<td>905 Need more granularity in EFI_RESET_TYPE to support platform specific resets</td>
<td>April 25, 2013</td>
</tr>
<tr>
<td>2.4</td>
<td>1052 EFI 2.4 Draft April 25th - Corrections to ARM sections</td>
<td>May 16, 2013</td>
</tr>
<tr>
<td>2.4</td>
<td>1050 2.4 Draft April 25 has missing text for ECR 1009</td>
<td>May 16, 2013</td>
</tr>
<tr>
<td>2.4</td>
<td>1049 2.4 Draft April 25 has missing text for ECR 1008</td>
<td>May 16, 2013</td>
</tr>
<tr>
<td>2.4</td>
<td>1048 Comment against UEFI 2.4 - NVMe related</td>
<td>May 16, 2013</td>
</tr>
<tr>
<td>2.4</td>
<td>1047 Comment on Feb 25th draft - fix alignment issue</td>
<td>May 16, 2013</td>
</tr>
<tr>
<td>2.4</td>
<td>1045 PCI OpROM Device List Changes to section 14.2</td>
<td>June 28, 2013</td>
</tr>
<tr>
<td>2.4</td>
<td>1044 Corrections to Mantis 1015, Interruptible driver diagnostics</td>
<td>May 16, 2013</td>
</tr>
<tr>
<td>2.4</td>
<td>1037 Add 2.4 to the system table version</td>
<td>May 16, 2013</td>
</tr>
<tr>
<td>2.4</td>
<td>1036 Comments on April 25 Draft</td>
<td>May 16, 2013</td>
</tr>
<tr>
<td>2.4</td>
<td>1033 HiilConfigAccess-&gt;ExtractConfig StatusCodes Errata</td>
<td>May 16, 2013</td>
</tr>
<tr>
<td>2.4</td>
<td>1032 HiilConfigRouting-&gt;ExtractConfig StatusCodes Errata</td>
<td>May 16, 2013</td>
</tr>
<tr>
<td>2.4</td>
<td>1031 NVMe subtype Conflict errata</td>
<td>April 25, 2013</td>
</tr>
<tr>
<td>2.4</td>
<td>1029 Method for delivery ofCapsule on disk; Method for reporting Capsule processing status</td>
<td>April 25, 2013</td>
</tr>
<tr>
<td>2.4</td>
<td>1026 (supersedes 993) Update to the AArch64 proposed Binding Change</td>
<td>April 25, 2013</td>
</tr>
<tr>
<td>2.4</td>
<td>1024 Clarification to the NVMe Device Path text descriptions</td>
<td>April 25, 2013</td>
</tr>
<tr>
<td>2.4</td>
<td>1023 Definition ofCapsule format to deliver update image to firmware management protocol</td>
<td>April 25, 2013</td>
</tr>
<tr>
<td>2.4</td>
<td>1022 adapter information protocol for NIC iSCSI and FCoE BootCapabilities and Current Boot Mode</td>
<td>April 25, 2013</td>
</tr>
<tr>
<td>2.4</td>
<td>1017 AIP Instance - FCOE SAN MAC Address</td>
<td>April 25, 2013</td>
</tr>
<tr>
<td>2.4</td>
<td>1016 AIP Instance - Image Update</td>
<td>April 25, 2013</td>
</tr>
<tr>
<td>2.4</td>
<td>1015 Interruptible driver diagnostics</td>
<td>April 25, 2013</td>
</tr>
<tr>
<td>2.4</td>
<td>1009 Enable hashes of Certificates to be used for revocation, and timestamp support</td>
<td>April 25, 2013</td>
</tr>
<tr>
<td>2.4</td>
<td>1008 New Random Number Generator / Entropy Protocol</td>
<td>April 25, 2013</td>
</tr>
<tr>
<td>2.4</td>
<td>1007 Create a new Security Technologies section to avoid Blurring with Secure Boot</td>
<td>April 25, 2013</td>
</tr>
<tr>
<td>2.4</td>
<td>1002 Timestamp Protocol</td>
<td>April 25, 2013</td>
</tr>
<tr>
<td>2.3.1D</td>
<td>996 EFI 2.0 version number still in the 2.3.1C spec</td>
<td>April 3, 2013</td>
</tr>
<tr>
<td>2.3.1D</td>
<td>995 CSA link Change</td>
<td>April 3, 2013</td>
</tr>
<tr>
<td>2.3.1D</td>
<td>994 Spec typos</td>
<td>April 3, 2013</td>
</tr>
<tr>
<td>2.3.1D</td>
<td>990 EFI_ATA_PASS_THRU need one Clarification if it supports ATAPI device</td>
<td>April 3, 2013</td>
</tr>
<tr>
<td>2.3.1D</td>
<td>989 Clarify hot-remove responsibility of a Bus Driver</td>
<td>April 3, 2013</td>
</tr>
<tr>
<td>2.3.1D</td>
<td>988 EFI_BLOCK_IO2_PROTOCOL Blocks Child from stopping while doing non-blocking I/O</td>
<td>April 3, 2013</td>
</tr>
<tr>
<td>2.3.1D</td>
<td>987 EFI_BLOCK_IO2_PROTOCOL has a Copy paste Bug describing the Token Parameter</td>
<td>April 3, 2013</td>
</tr>
<tr>
<td>2.3.1D</td>
<td>980 Errata on SNP Media detect</td>
<td>April 3, 2013</td>
</tr>
<tr>
<td>2.3.1D</td>
<td>978 Error Return Indicates Capsule requires Boot Services</td>
<td>April 3, 2013</td>
</tr>
<tr>
<td>2.3.1D</td>
<td>977 missing statement</td>
<td>April 3, 2013</td>
</tr>
<tr>
<td>Revision</td>
<td>Mantis Number / Description</td>
<td>Date</td>
</tr>
<tr>
<td>---------</td>
<td>-----------------------------</td>
<td>---------------</td>
</tr>
<tr>
<td>2.3.1D</td>
<td>976 BrowserCallback text update to description</td>
<td>April 3, 2013</td>
</tr>
<tr>
<td>2.3.1D</td>
<td>975 UNDI errata to add missing memory type definitions</td>
<td>April 3, 2013</td>
</tr>
<tr>
<td>2.3.1D</td>
<td>974 UNDI IncorrectCPB function names ECR</td>
<td>April 3, 2013</td>
</tr>
<tr>
<td>2.3.1D</td>
<td>973 UNDI Mem_Map()Clarification</td>
<td>April 3, 2013</td>
</tr>
<tr>
<td>2.3.1D</td>
<td>972 ISCSI DHCP6Boot</td>
<td>April 3, 2013</td>
</tr>
<tr>
<td>2.3.1D</td>
<td>971 typo</td>
<td>April 3, 2013</td>
</tr>
<tr>
<td>2.3.1D</td>
<td>970 Typo section 28.3.8.3.41 EFI_IFR_MODAL_TAG</td>
<td>April 3, 2013</td>
</tr>
<tr>
<td>2.3.1D</td>
<td>965 File IO Async extension</td>
<td>April 3, 2013</td>
</tr>
<tr>
<td>2.3.1D</td>
<td>962 Remove 2.3 table revision number</td>
<td>April 3, 2013</td>
</tr>
<tr>
<td>2.3.1D</td>
<td>960 Typo in netboot6 description</td>
<td>April 3, 2013</td>
</tr>
<tr>
<td>2.3.1D</td>
<td>959 InstallAcpiTable() does not say what to do when an attempt is made to install a duplicate table</td>
<td>April 3, 2013</td>
</tr>
<tr>
<td>2.3.1D</td>
<td>955 Clearing The Platform Key Errata</td>
<td>April 3, 2013</td>
</tr>
<tr>
<td>2.3.1D</td>
<td>954 LoadImage Errata</td>
<td>April 3, 2013</td>
</tr>
<tr>
<td>2.3.1D</td>
<td>953 Need text definitions for Device Path Media Type Subtype 6/7</td>
<td>April 3, 2013</td>
</tr>
<tr>
<td>2.3.1D</td>
<td>952 Clarification of requirements to update timestamp associated with authenticated variable</td>
<td>April 3, 2013</td>
</tr>
<tr>
<td>2.3.1D</td>
<td>950 IndeterminateBehavior for attribute modifications may Cause security issues</td>
<td>April 3, 2013</td>
</tr>
<tr>
<td>2.3.1D</td>
<td>949 PCI IO.GetBarAttributes needs adjustment -- Address Space Granularity field</td>
<td>April 3, 2013</td>
</tr>
<tr>
<td>2.3.1D</td>
<td>944 Errata - Replace RFC reference</td>
<td>April 3, 2013</td>
</tr>
<tr>
<td>2.3.1D</td>
<td>943 Errata - Proposed updates to required interfaces inChapter 2.6</td>
<td>April 3, 2013</td>
</tr>
<tr>
<td>2.3.1D</td>
<td>942 ExportConfig() description does not make sense</td>
<td>April 3, 2013</td>
</tr>
<tr>
<td>2.3.1D</td>
<td>941 Add OEM StatusCode ranges to EFI StatusCode Ranges Table</td>
<td>April 3, 2013</td>
</tr>
<tr>
<td>2.3.1D</td>
<td>938 InstallMultipleProtocolInterface() is missing StatusCode Returned values</td>
<td>April 3, 2013</td>
</tr>
<tr>
<td>2.3.1D</td>
<td>935 Clarify Chaining requirements with regards to the Platform Key</td>
<td>April 3, 2013</td>
</tr>
<tr>
<td>2.3.1D</td>
<td>934 Missing Figures and typos</td>
<td>April 3, 2013</td>
</tr>
<tr>
<td>2.3.1D</td>
<td>930 Clarify usage of EFI Variable Varstores in HII</td>
<td>April 3, 2013</td>
</tr>
<tr>
<td>2.3.1D</td>
<td>928 Best Matching Language algorithm</td>
<td>April 3, 2013</td>
</tr>
<tr>
<td>2.3.1D</td>
<td>926 UEFI Image Verification Clarification</td>
<td>April 3, 2013</td>
</tr>
<tr>
<td>2.3.1D</td>
<td>924 New ErrorCode to handle reporting of IPV4 duplicate address detection</td>
<td>April 3, 2013</td>
</tr>
<tr>
<td>2.3.1D</td>
<td>1021 ATA_PASS_THRU on ATAPI device handle</td>
<td>April 3, 2013</td>
</tr>
<tr>
<td>2.3.1D</td>
<td>1020 Clarify HII variable store definitions</td>
<td>April 3, 2013</td>
</tr>
<tr>
<td>2.3.1D</td>
<td>1019 Alignment Requirements Clarification</td>
<td>April 3, 2013</td>
</tr>
<tr>
<td>2.3.1D</td>
<td>1018 HII Font Errata</td>
<td>April 3, 2013</td>
</tr>
<tr>
<td>2.3.1D</td>
<td>1013 HII Errata</td>
<td>April 3, 2013</td>
</tr>
<tr>
<td>2.3.1D</td>
<td>1012 Touchup to text of GPT</td>
<td>April 3, 2013</td>
</tr>
<tr>
<td>2.3.1D</td>
<td>1011 Typo regarding Debug Port in UEFI Spec</td>
<td>April 3, 2013</td>
</tr>
<tr>
<td>2.3.1D</td>
<td>1003 Missing “” in section 11.7</td>
<td>April 3, 2013</td>
</tr>
<tr>
<td>2.3.1D</td>
<td>1000 Clarification to the IFR_REF4 opcode</td>
<td>April 3, 2013</td>
</tr>
<tr>
<td>2.3.1C</td>
<td>921 Length of IPv6 Device Path is incorrect</td>
<td>June 13, 2012</td>
</tr>
<tr>
<td>Revision</td>
<td>Mantis Number / Description</td>
<td>Date</td>
</tr>
<tr>
<td>----------</td>
<td>---------------------------------------------------------------------------------------------</td>
<td>------------</td>
</tr>
<tr>
<td>2.3.1C</td>
<td>917 UNDI drive does not need to be initialized as runtime driver</td>
<td>June 13, 2012</td>
</tr>
<tr>
<td>2.3.1C</td>
<td>915 For x64, Change Floating Point Default Configuration to Double-Extended Precision</td>
<td>June 13, 2012</td>
</tr>
<tr>
<td>2.3.1C</td>
<td>914 Error Descriptor Reset Flag Clarification</td>
<td>June 13, 2012</td>
</tr>
<tr>
<td>2.3.1C</td>
<td>913 Enum definition does not match what ourCurrentCompilers implement.</td>
<td>June 13, 2012</td>
</tr>
<tr>
<td>2.3.1C</td>
<td>912 UEFI 2.3.1 Type</td>
<td>June 13, 2012</td>
</tr>
<tr>
<td>2.3.1C</td>
<td>909 Update to returnCodes for AllocatePool / AllocatePages</td>
<td>June 13, 2012</td>
</tr>
<tr>
<td>2.3.1C</td>
<td>907 iSCSI Device Path error</td>
<td>June 13, 2012</td>
</tr>
<tr>
<td>2.3.1C</td>
<td>882 Indications Variable - OS/FW feature &amp; Capability Communication</td>
<td>June 13, 2012</td>
</tr>
<tr>
<td>2.3.1C</td>
<td>882 Indications Variable - OS/FW feature &amp; Capability Communication</td>
<td>June 13, 2012</td>
</tr>
<tr>
<td>2.3.1C</td>
<td>874 Provide a mechanism for providing keys in setup mode</td>
<td>June 13, 2012</td>
</tr>
<tr>
<td>2.3.1C</td>
<td>873 PXEBootCSA Type definition Cleanup</td>
<td>June 13, 2012</td>
</tr>
<tr>
<td>2.3.1B</td>
<td>896 StartImage and ConnectController returnCodes</td>
<td>April 10, 2012</td>
</tr>
<tr>
<td>2.3.1B</td>
<td>893 SMM Communication ACPI Table Update</td>
<td>April 10, 2012</td>
</tr>
<tr>
<td>2.3.1B</td>
<td>891 Component Name Protocol References</td>
<td>April 10, 2012</td>
</tr>
<tr>
<td>2.3.1B</td>
<td>890 DriveConfiguration Protocol Phantom</td>
<td>April 10, 2012</td>
</tr>
<tr>
<td>2.3.1B</td>
<td>888 typo in EFI_USB_HC Protocol</td>
<td>April 10, 2012</td>
</tr>
<tr>
<td>2.3.1B</td>
<td>887 union is declared twice in same section</td>
<td>April 10, 2012</td>
</tr>
<tr>
<td>2.3.1B</td>
<td>885 Errata in the GPT Table structure</td>
<td>April 10, 2012</td>
</tr>
<tr>
<td>2.3.1B</td>
<td>884 EFI_BOOT_KEY_DATA relies on implementation-defined behavior</td>
<td>April 10, 2012</td>
</tr>
<tr>
<td>2.3.1B</td>
<td>881 netboot6 - multicast versus unicast</td>
<td>April 10, 2012</td>
</tr>
<tr>
<td>2.3.1B</td>
<td>880 netboot6 Clarification/errata</td>
<td>April 10, 2012</td>
</tr>
<tr>
<td>2.3.1B</td>
<td>879 Reference to unsupported specification in SCSIChapter (14.1)</td>
<td>April 10, 2012</td>
</tr>
<tr>
<td>2.3.1B</td>
<td>878 Updated HII &quot;Selected Form&quot; Behaviors to Reflect NewCallback Results</td>
<td>April 10, 2012</td>
</tr>
<tr>
<td>2.3.1B</td>
<td>877 Table Checksum updateBy the ACPI_TABLE_PROTOCOL.InstallAcpiTable</td>
<td>April 10, 2012</td>
</tr>
<tr>
<td>2.3.1B</td>
<td>876 To Clarify EDID_OVERRIDE attribute definitions and expected operations</td>
<td>April 10, 2012</td>
</tr>
<tr>
<td>2.3.1B</td>
<td>873 Section 9.3.7 incorrectly assumes that all uses of BBS device paths are non-UEFI</td>
<td>April 10, 2012</td>
</tr>
<tr>
<td>2.3.1B</td>
<td>872 Change to SIMPLE_TEXT_INPUT_EX_PROTOCOL.RegisterKeyNotify/UnregisterKeyNotify</td>
<td>April 10, 2012</td>
</tr>
<tr>
<td>2.3.1B</td>
<td>871 Typo in InstallMultipleProtocolInterfaces</td>
<td>April 10, 2012</td>
</tr>
<tr>
<td>2.3.1B</td>
<td>870 Clarify FrameBufferSize definition under EFI_GRAPHICS_OUTPUT_PROTOCOL_MODE struct</td>
<td>April 10, 2012</td>
</tr>
<tr>
<td>2.3.1B</td>
<td>869 Reference to FIPS 180 in Chapter 27.3 is obsolete and incorrect</td>
<td>April 10, 2012</td>
</tr>
<tr>
<td>2.3.1B</td>
<td>867 Clarify requirement for use of EFI_HASH_SERVICE_BINDING_PROTOCOL</td>
<td>April 10, 2012</td>
</tr>
<tr>
<td>2.3.1B</td>
<td>866 PK, KEK, db, dbx relations Clarification</td>
<td>April 10, 2012</td>
</tr>
<tr>
<td>2.3.1B</td>
<td>865 Modify Protective MBR_BOOT_INDICATOR definition</td>
<td>April 10, 2012</td>
</tr>
<tr>
<td>2.3.1B</td>
<td>864 Typo in Question-Level Validation section</td>
<td>April 10, 2012</td>
</tr>
<tr>
<td>2.3.1B</td>
<td>863 Attributes of the Globally Defined Variables</td>
<td>April 10, 2012</td>
</tr>
<tr>
<td>2.3.1B</td>
<td>862 User identity typo</td>
<td>April 10, 2012</td>
</tr>
<tr>
<td>2.3.1B</td>
<td>861 Globally Defined Variables Errata</td>
<td>April 10, 2012</td>
</tr>
<tr>
<td>2.3.1B</td>
<td>858 Superfluous and incorrect image hash description</td>
<td>April 10, 2012</td>
</tr>
<tr>
<td>Revision</td>
<td>Mantis Number / Description</td>
<td>Date</td>
</tr>
<tr>
<td>-----------</td>
<td>---------------------------------------------------------------------------------------------</td>
<td>------------------</td>
</tr>
<tr>
<td>2.3.1B</td>
<td>857 Absolute pointer typo</td>
<td>April 10, 2012</td>
</tr>
<tr>
<td>2.3.1B</td>
<td>855 Clarification of UEFI driver signing/Code definitions</td>
<td>April 10, 2012</td>
</tr>
<tr>
<td>2.3.1B</td>
<td>853 The EFI_HASH_PROTOCOL.Hash() description needs Clarification on padding responsibilities</td>
<td>April 10, 2012</td>
</tr>
<tr>
<td>2.3.1B</td>
<td>852 Various EFI_IFR_REFRESH_ID errata</td>
<td>April 10, 2012</td>
</tr>
<tr>
<td>2.3.1B</td>
<td>851 For EFI_IFR_REFRESH opcode, Clarify RefreshInterval = 0 means no auto-refresh.</td>
<td>April 10, 2012</td>
</tr>
<tr>
<td>2.3.1B</td>
<td>850 Clarification of responsibility for array allocation in EFI_HASH_PROTOCOL</td>
<td>April 10, 2012</td>
</tr>
<tr>
<td>2.3.1B</td>
<td>849 EFI_IFR_MODAL_TAG_OP is also valid under EFI_IFR_FORM_MAP_OP</td>
<td>April 10, 2012</td>
</tr>
<tr>
<td>2.3.1B</td>
<td>848 Clarification of semantics of SecureBoot variable</td>
<td>April 10, 2012</td>
</tr>
<tr>
<td>2.3.1B</td>
<td>847 When enrolling a PK, the platform shall not require a reboot to leave SetupMode</td>
<td>April 10, 2012</td>
</tr>
<tr>
<td>2.3.1B</td>
<td>845 EFI_SCSI_PASS_THRU_PROTOCOL replacement</td>
<td>April 10, 2012</td>
</tr>
<tr>
<td>2.3.1B</td>
<td>842 Text to explain how the UEFI revision is referred</td>
<td>April 10, 2012</td>
</tr>
<tr>
<td>2.3.1B</td>
<td>836 StructureComment for EFI_IFR_TYPE_VALUE references unknown value type.</td>
<td>April 10, 2012</td>
</tr>
<tr>
<td>2.3.1B</td>
<td>828 Network Driver Options</td>
<td>April 10, 2012</td>
</tr>
<tr>
<td>2.3.1B</td>
<td>826 Comments against Mantis 790</td>
<td>April 10, 2012</td>
</tr>
<tr>
<td>2.3.1B</td>
<td>825 DMTF SMCLP errata</td>
<td>April 10, 2012</td>
</tr>
<tr>
<td>2.3.1B</td>
<td>819 Mantis 715 was not fully implemented</td>
<td>April 10, 2012</td>
</tr>
<tr>
<td>2.3.1B</td>
<td>812 Errata – DUID-UUID usage</td>
<td>April 10, 2012</td>
</tr>
<tr>
<td>2.3.1B</td>
<td>809 Errata – Messaging Device Path Clarification</td>
<td>April 10, 2012</td>
</tr>
<tr>
<td>2.3.1B</td>
<td>808 Errata – Boot File URL</td>
<td>April 10, 2012</td>
</tr>
<tr>
<td>2.3.1B</td>
<td>807 Give specific TPL rules to Stall()Boot services</td>
<td>April 10, 2012</td>
</tr>
<tr>
<td>2.3.1B</td>
<td>771 SHA1 and MD5 references</td>
<td>April 10, 2012</td>
</tr>
<tr>
<td>2.3.1A</td>
<td>Minor Corrections in toes to tickets 772, 785, 794, 804, also formatting Correction for _WIN_CERTIFICATE_UEFI_GUID typedef’s parameters</td>
<td>September 7, 2011</td>
</tr>
<tr>
<td>2.3.1A</td>
<td>820 Driver Health Needs to have Mantis 0000169 implemented</td>
<td>August 17, 2011</td>
</tr>
<tr>
<td>2.3.1A</td>
<td>819 ECR715 was not fully implemented</td>
<td>August 17, 2011</td>
</tr>
<tr>
<td>2.3.1A</td>
<td>806 Text update to Driver Health Description - Clarify role of user interaction</td>
<td>August 17, 2011</td>
</tr>
<tr>
<td>2.3.1A</td>
<td>805 Correct Wrong Palette Information in 28.3.7.2.3 example</td>
<td>August 17, 2011</td>
</tr>
<tr>
<td>2.3.1A</td>
<td>804 Clarify Contraints and alternatives when enrolling PK, KeK, db or dbx keys</td>
<td>August 17, 2011</td>
</tr>
<tr>
<td>2.3.1A</td>
<td>803 Fix AcpiExp device node text description.</td>
<td>August 17, 2011</td>
</tr>
<tr>
<td>2.3.1A</td>
<td>801 Clarify IFR Opcode Summary and Description #4</td>
<td>August 17, 2011</td>
</tr>
<tr>
<td>2.3.1A</td>
<td>800 Clarify IFR Opcode Summary and Description #3</td>
<td>August 17, 2011</td>
</tr>
<tr>
<td>2.3.1A</td>
<td>797 Clarify IFR Opcode Summary and Description #2</td>
<td>August 17, 2011</td>
</tr>
<tr>
<td>2.3.1A</td>
<td>796 Clarify IFR Opcode Summary and Description #1</td>
<td>August 17, 2011</td>
</tr>
<tr>
<td>2.3.1A</td>
<td>795 Typo in ReadKeyStrokeEx()</td>
<td>August 17, 2011</td>
</tr>
<tr>
<td>2.3.1A</td>
<td>794 Incomplete text describing Clearing of Platform Key</td>
<td>August 17, 2011</td>
</tr>
<tr>
<td>2.3.1A</td>
<td>793 Inconsistent wording about RemainingDevicePath</td>
<td>August 17, 2011</td>
</tr>
<tr>
<td>2.3.1A</td>
<td>790 Add warning to ReadKeyStrokeEx for partial key press</td>
<td>August 17, 2011</td>
</tr>
<tr>
<td>2.3.1A</td>
<td>789 Clarify HII opcode definition</td>
<td>August 17, 2011</td>
</tr>
<tr>
<td>Revision</td>
<td>Mantis Number / Description</td>
<td>Date</td>
</tr>
<tr>
<td>----------</td>
<td>---------------------------------------------------------------------------------------------</td>
<td>----------------</td>
</tr>
<tr>
<td>2.3.1A</td>
<td>788 SasEx entry in Table 86-Device Node Table Contains optional Reserved entry that does not exist in device path</td>
<td>August 17, 2011</td>
</tr>
<tr>
<td>2.3.1A</td>
<td>786 PCI I/O Dual AddressCycle attribute Clarification</td>
<td>August 17, 2011</td>
</tr>
<tr>
<td>2.3.1A</td>
<td>785 Allowing more general use of UEFI 2.3.1 Variable time-based authentication</td>
<td>August 17, 2011</td>
</tr>
<tr>
<td>2.3.1A</td>
<td>780 Errata in returnCode descriptions</td>
<td>August 17, 2011</td>
</tr>
<tr>
<td>2.3.1A</td>
<td>778 EFI_HII_CONFIG_ACCESS_PROTOCOL.CallBack() Errata</td>
<td>August 17, 2011</td>
</tr>
<tr>
<td>2.3.1A</td>
<td>777 Specified signature sizes incorrect in Section 27.6.1</td>
<td>August 17, 2011</td>
</tr>
<tr>
<td>2.3.1A</td>
<td>776 Clarify computation of EFI_VARIABLE_AUTHENTICATION_2 hash value</td>
<td>August 17, 2011</td>
</tr>
<tr>
<td>2.3.1A</td>
<td>774 Define EFI_BLOCK_IO_PROTOCOL_REVISION3</td>
<td>August 17, 2011</td>
</tr>
<tr>
<td>2.3.1A</td>
<td>773 Clarify the value for opcode EFI_IFR_REFRESH_ID_OP</td>
<td>August 17, 2011</td>
</tr>
<tr>
<td>2.3.1A</td>
<td>772 Definition of EFI_IMAGE_SECURITY_DATABASE_GUID incorrect</td>
<td>August 17, 2011</td>
</tr>
<tr>
<td>2.3.1A</td>
<td>770 Remove references to UEFI 2.1 spec</td>
<td>August 17, 2011</td>
</tr>
<tr>
<td>2.3.1A</td>
<td>767 The ReadBlocks function for BlockIO and BlockIO2 need synchronization</td>
<td>August 17, 2011</td>
</tr>
<tr>
<td>2.3.1A</td>
<td>765 ECR to limit the hash and encryption algorithms used with PKCSCertificates</td>
<td>April 5, 2011</td>
</tr>
<tr>
<td>2.3.1A</td>
<td>762 DevicePath in the Image Execution Information Table.</td>
<td>April 5, 2011</td>
</tr>
<tr>
<td>2.3.1A</td>
<td>761 Table 195. Information for Types of Storage</td>
<td>April 5, 2011</td>
</tr>
<tr>
<td>2.3.1A</td>
<td>760 Suggested Changes to 2.3.1 final draft spec</td>
<td>April 5, 2011</td>
</tr>
<tr>
<td>2.3.1A</td>
<td>759 UEFI Errata - wincerts for rest of hash algorithms</td>
<td>April 5, 2011</td>
</tr>
<tr>
<td>2.3.1A</td>
<td>755 Errata in Legacy MBR table and Legacy MBR GUID</td>
<td>April 5, 2011</td>
</tr>
<tr>
<td>2.3.1A</td>
<td>754 USB timeout parameter mismatch.</td>
<td>April 5, 2011</td>
</tr>
<tr>
<td>2.3.1A</td>
<td>751 Fix USB HC2 erroneous references to IsSlowDevice</td>
<td>March 11, 2011</td>
</tr>
<tr>
<td>2.3.1A</td>
<td>750 Fix section 27.2.5 &quot;related definitions&quot; re: RSA public key exponent</td>
<td>March 11, 2011</td>
</tr>
<tr>
<td>2.3.1A</td>
<td>749 Fix Table 10 (Global Variables) WithCorrect Attributes</td>
<td>March 11, 2011</td>
</tr>
<tr>
<td>2.3.1A</td>
<td>748 Clarify Standard GUID Text Representation</td>
<td>March 11, 2011</td>
</tr>
<tr>
<td>2.3.1A</td>
<td>744 ProcessorContext information structure definition notClear</td>
<td>March 11, 2011</td>
</tr>
<tr>
<td>2.3.1A</td>
<td>741 Errata: Corrected text for section 7.2.1.4 step 7</td>
<td>March 11, 2011</td>
</tr>
<tr>
<td>2.3.1A</td>
<td>740 Errata: signatureheadersize inconsistency Corrections</td>
<td>April 6, 2011</td>
</tr>
<tr>
<td>2.3.1A</td>
<td>736 Insert SMM Communication ACPI Table and related data structures to the UEFI Specification</td>
<td>April 5, 2011</td>
</tr>
<tr>
<td>2.3.1A</td>
<td>735 Clarification on Tape Header Format</td>
<td>March 11, 2011</td>
</tr>
<tr>
<td>2.3.1A</td>
<td>734 SecureBoot variable</td>
<td>April 5, 2011</td>
</tr>
<tr>
<td>2.3.1A</td>
<td>733 Errata: 27.6.1 signatureheadersize definition</td>
<td>March 11, 2011</td>
</tr>
<tr>
<td>2.3.1A</td>
<td>732 Amendment to Mantis 711: section 7.2.1.6</td>
<td>March 11, 2011</td>
</tr>
<tr>
<td>2.3.1A</td>
<td>729 Errata: Clarification of Microsoft references in appendix Q</td>
<td>March 11, 2011</td>
</tr>
<tr>
<td>2.3.1A</td>
<td>728 Netboot 6 errata - DUID-UUID</td>
<td>March 11, 2011</td>
</tr>
<tr>
<td>2.3.1A</td>
<td>727 Errata on returnCode for User Info Identity policy record</td>
<td>March 11, 2011</td>
</tr>
<tr>
<td>2.3.1A</td>
<td>726 Errata: clean-up of EFI_DHCP4_TRANSMIT_RECEIVE_TOKEN definition</td>
<td>March 11, 2011</td>
</tr>
<tr>
<td>2.3.1A</td>
<td>724 SetVariable Update 2</td>
<td>March 11, 2011</td>
</tr>
<tr>
<td>2.3.1A</td>
<td>723 User Identification (UID) Errata – EFI User Manager Notify &amp; EnrollClarification</td>
<td>April 5, 2011</td>
</tr>
<tr>
<td>Revision</td>
<td>Mantis Number / Description</td>
<td>Date</td>
</tr>
<tr>
<td>----------</td>
<td>----------------------------</td>
<td>------------</td>
</tr>
<tr>
<td>2.3.1</td>
<td>722 User Identification (UID) Errata –Credential Provider EnrollClarification</td>
<td>April 5, 2011</td>
</tr>
<tr>
<td>2.3.1</td>
<td>721 User Identification (UID) Errata – SetInfoClarification</td>
<td>March 11, 2011</td>
</tr>
<tr>
<td>2.3.1</td>
<td>720 User Identification (UID) Errata –Credential Provider EnrollClarification</td>
<td>March 11, 2011</td>
</tr>
<tr>
<td>2.3.1</td>
<td>716 EFI_EXT_SCSI_PASS_THRU_PROTOCOL.GetNextTarget() IN OUT parameter Target input value shall Be 0xFFs</td>
<td>March 11, 2011</td>
</tr>
<tr>
<td>2.3.1</td>
<td>715 CPER Record and section fieldClarification</td>
<td>March 11, 2011</td>
</tr>
<tr>
<td>2.3.1</td>
<td>713 Remove the errata revision from the EFI_IFR_VERSION format.</td>
<td>March 11, 2011</td>
</tr>
<tr>
<td>2.3.1</td>
<td>711 SetVariable Update</td>
<td>March 11, 2011</td>
</tr>
<tr>
<td>2.3.1</td>
<td>709 NewCallback() Action Requests Related To Individual Forms.</td>
<td>Feb. 3, 2011</td>
</tr>
<tr>
<td>2.3.1</td>
<td>708 Errata (non-blocking BLOCK IO)</td>
<td>April 5, 2011</td>
</tr>
<tr>
<td>2.3.1</td>
<td>707 Errata revision in the EFI_IFR_VERSION format</td>
<td>Feb. 3, 2011</td>
</tr>
<tr>
<td>2.3.1</td>
<td>705 REPC signature definition still Confusing</td>
<td>Feb. 3, 2011</td>
</tr>
<tr>
<td>2.3.1</td>
<td>704 Unload() definition is wrong</td>
<td>Feb. 3, 2011</td>
</tr>
<tr>
<td>2.3.1</td>
<td>702 Clarifications on Variable Storage for Questions</td>
<td>Feb. 3, 2011</td>
</tr>
<tr>
<td>2.3.1</td>
<td>696 Update System Table with this new #define for EFI_SYSTEM_TABLE_REVISION</td>
<td>Feb. 3, 2011</td>
</tr>
<tr>
<td>2.3.1</td>
<td>695 Add Port Ownership probing</td>
<td>Feb. 3, 2011</td>
</tr>
<tr>
<td>2.3.1</td>
<td>687 Update System Table with this new #define for 2.3.1</td>
<td>Jan. 17, 2011</td>
</tr>
<tr>
<td>2.3.1</td>
<td>686 HII - Clarify FormsBrowser 'standard' user intera ctions.</td>
<td>Feb. 3, 2011</td>
</tr>
<tr>
<td>2.3.1</td>
<td>685 HII - New op-code to enable event initiated refresh ofBrowserContext data</td>
<td>Feb. 3, 2011</td>
</tr>
<tr>
<td>2.3.1</td>
<td>682 [UCST] Modal Form</td>
<td>Feb. 3, 2011</td>
</tr>
<tr>
<td>2.3.1</td>
<td>681 Typo: Pg. 56</td>
<td>Jan. 17, 2011</td>
</tr>
<tr>
<td>2.3.1</td>
<td>680 Netboot6 handleClarification</td>
<td>Jan. 17, 2011</td>
</tr>
<tr>
<td>2.3.1</td>
<td>679 UEFI Authenticated Variable &amp; Signature Database Updates</td>
<td>Jan. 17, 2011</td>
</tr>
<tr>
<td>2.3.1</td>
<td>678 Section 27.6.2: Imagehash reference needs to Be removed</td>
<td>Jan. 17, 2011</td>
</tr>
<tr>
<td>2.3.1</td>
<td>677 Section 27.2.5 &amp; 27.6.1: Typo in X509 Signature Type</td>
<td>Jan. 17, 2011</td>
</tr>
<tr>
<td>2.3.1</td>
<td>674 Section 3.2: Missing variable type for SetupMode variable</td>
<td>Jan. 17, 2011</td>
</tr>
<tr>
<td>2.3.1</td>
<td>671 Errata: USB device path example is incorrect</td>
<td>Jan. 17, 2011</td>
</tr>
<tr>
<td>2.3.1</td>
<td>668 LUN implementations are not Consistent</td>
<td>Feb. 3, 2011</td>
</tr>
<tr>
<td>2.3.1</td>
<td>661 USB 3.0 Updates</td>
<td>Oct. 29, 2010</td>
</tr>
<tr>
<td>2.3.1</td>
<td>645 Non-blocking interface for BLOCK oriented devices (BLOCK_IO_EX transition to BLOCK_IO_2)</td>
<td>Oct. 29, 2010</td>
</tr>
<tr>
<td>2.3.1</td>
<td>634 FormsBrowser DefaultBehavior</td>
<td>Jan. 17, 2011</td>
</tr>
<tr>
<td>2.3.1</td>
<td>634 FormsBrowser DefaultBehavior</td>
<td>Oct. 29, 2010</td>
</tr>
<tr>
<td>2.3.1</td>
<td>616 Security ProtocolCommand to support encrypted HDD</td>
<td>Jan. 17, 2011</td>
</tr>
<tr>
<td>2.3.1</td>
<td>616 Security ProtocolCommand to support encrypted HDD</td>
<td>Oct. 29, 2010</td>
</tr>
<tr>
<td>2.3.1</td>
<td>612 UEFI system Partition FAT32 data Region Alignment</td>
<td>Oct. 29, 2010</td>
</tr>
<tr>
<td>2.3.1</td>
<td>484 Key Management Service Protocol</td>
<td>Oct. 28, 2010</td>
</tr>
<tr>
<td>2.3.1</td>
<td>484 Key Management Service (KMS) Protocol</td>
<td>Oct. 29, 2010</td>
</tr>
<tr>
<td>Revision</td>
<td>Mantis Number / Description</td>
<td>Date</td>
</tr>
<tr>
<td>----------</td>
<td>----------------------------</td>
<td>---------------</td>
</tr>
<tr>
<td>2.3.1</td>
<td>478 (REVISIT) Update to ALTCFG references</td>
<td>March 11, 2011</td>
</tr>
<tr>
<td>2.3 D</td>
<td>667 Clarification to the UEFIConfiguration Table definition</td>
<td>Oct. 28, 2010</td>
</tr>
<tr>
<td>2.3 D</td>
<td>664 Appendix update for IPV6 networkBoot</td>
<td>Oct. 28, 2010</td>
</tr>
<tr>
<td>2.3 D</td>
<td>663 Update ARM PlatformBinding to allow OS loader to assume unaligned access support is enabled</td>
<td>Nov. 10, 2010</td>
</tr>
<tr>
<td>2.3 D</td>
<td>662 ARM ABI errata</td>
<td>Oct. 28, 2010</td>
</tr>
<tr>
<td>2.3 D</td>
<td>659 Clarify section length definition in the error record</td>
<td>Oct. 28, 2010</td>
</tr>
<tr>
<td>2.3 D</td>
<td>653 Errata to the Appendix N (Common Platform Error Record)</td>
<td>Oct. 28, 2010</td>
</tr>
<tr>
<td>2.3 D</td>
<td>652 Clarification to the TimeZone value usage</td>
<td>Oct. 28, 2010</td>
</tr>
<tr>
<td>2.3 D</td>
<td>651 update to IPSec for tunnel mode support</td>
<td>Oct. 28, 2010</td>
</tr>
<tr>
<td>2.3 D</td>
<td>650 networking support errata</td>
<td>Oct. 28, 2010</td>
</tr>
<tr>
<td>2.3 D</td>
<td>638 Add facility for dynamic IFR dynamicCross-references</td>
<td>Oct. 28, 2010</td>
</tr>
<tr>
<td>2.3 D</td>
<td>538 IPV6 PXE</td>
<td>Oct. 28, 2010</td>
</tr>
<tr>
<td>2.3 C</td>
<td>640 String ReferenceCleanup</td>
<td>July 14, 2010</td>
</tr>
<tr>
<td>2.3 C</td>
<td>639 Callback() does not describe FORM_OPEN/FORM_CLOSEBehavior</td>
<td>July 14, 2010</td>
</tr>
<tr>
<td>2.3 C</td>
<td>637 Clarification for Date/Time Question usage in IFR expressions.</td>
<td>July 14, 2010</td>
</tr>
<tr>
<td>2.3 C</td>
<td>636 Mistaken Reference to &quot;Date&quot; inside ofBoolean question description</td>
<td>July 14, 2010</td>
</tr>
<tr>
<td>2.3 C</td>
<td>635 Missing GUID label forConfig Access protocol</td>
<td>July 14, 2010</td>
</tr>
<tr>
<td>2.3 C</td>
<td>633 Explicitly Specify ACPI Table Signature Format</td>
<td>July 14, 2010</td>
</tr>
<tr>
<td>2.3 C</td>
<td>632 ClarifyBlock IO ReadBlocks and WriteBlocks functions handling of media stateChange events</td>
<td>July 14, 2010</td>
</tr>
<tr>
<td>2.3 C</td>
<td>625 Minor typo in surrogateCharacter description section</td>
<td>July 14, 2010</td>
</tr>
<tr>
<td>2.3 C</td>
<td>622 Identify() function errata</td>
<td>July 14, 2010</td>
</tr>
<tr>
<td>2.3 C</td>
<td>621 Typos in an EFI_HII_CONFIG_ACCESS_PROTOCOL.Callback() member</td>
<td>July 14, 2010</td>
</tr>
<tr>
<td>2.3 C</td>
<td>620 Clarification of need for Path MTU support for IPV4 and IPV6</td>
<td>July 14, 2010</td>
</tr>
<tr>
<td>2.3 C</td>
<td>613 PAUSE Key</td>
<td>July 14, 2010</td>
</tr>
<tr>
<td>2.3 C</td>
<td>611 LanguageCorrection requested for InstallProtocolInterface() and InstallConfigurationTable(), Ref# S83</td>
<td>July 14, 2010</td>
</tr>
<tr>
<td>2.3 C</td>
<td>610 RSA data structureClarification</td>
<td>July 14, 2010</td>
</tr>
<tr>
<td>2.3 C</td>
<td>609 StartImage returnCode update</td>
<td>July 14, 2010</td>
</tr>
<tr>
<td>2.3 C</td>
<td>583 How do we know an EFI_HANDLE is Valid/Invalid</td>
<td>July 14, 2010</td>
</tr>
<tr>
<td>2.3 C</td>
<td>508 Update networking references, incl ipv6</td>
<td>July 14, 2010</td>
</tr>
<tr>
<td>2.3 B</td>
<td>608 more media detect Clean-up</td>
<td>Feb. 24, 2010</td>
</tr>
<tr>
<td>2.3 B</td>
<td>605 Clarify user identity Find API</td>
<td>Feb. 24, 2010</td>
</tr>
<tr>
<td>2.3 B</td>
<td>601 UNDI update as part of media detectChanges</td>
<td>Feb. 24, 2010</td>
</tr>
<tr>
<td>2.3 B</td>
<td>600 Update toConfigAccess/ConfigRouting</td>
<td>Feb. 24, 2010</td>
</tr>
<tr>
<td>Revision</td>
<td>Mantis Number / Description</td>
<td>Date</td>
</tr>
<tr>
<td>----------</td>
<td>--------------------------------------------------------------------------------------------</td>
<td>------------</td>
</tr>
<tr>
<td>2.3B</td>
<td>598 ARP is only an IPV4 Concept.</td>
<td>Feb. 24, 2010</td>
</tr>
<tr>
<td>2.3B</td>
<td>590 Media detectClean-up</td>
<td>Feb. 24, 2010</td>
</tr>
<tr>
<td>2.3B</td>
<td>589 Device path representation of IPv4/v6 text</td>
<td>Feb. 24, 2010</td>
</tr>
<tr>
<td>2.3B</td>
<td>588 UEFI User Identity - ReturnCodes</td>
<td>Feb. 24, 2010</td>
</tr>
<tr>
<td>2.3B</td>
<td>587 UEFI User Identity - NamingConsistency</td>
<td>Feb. 24, 2010</td>
</tr>
<tr>
<td>2.3B</td>
<td>586 Clarification of PXE2.1 specification for IPV4 interoperability issues</td>
<td>Feb. 24, 2010</td>
</tr>
<tr>
<td>2.3B</td>
<td>585 Errata to EFI_IFR_SET op-code</td>
<td>Feb. 24, 2010</td>
</tr>
<tr>
<td>2.3B</td>
<td>584 EFI_PXE_BASE_CODE_DHCPV6_PACKET missing for pxeBc protocol</td>
<td>Feb. 24, 2010</td>
</tr>
<tr>
<td>2.3B</td>
<td>583 How do we know an EFI_HANDLE is Valid/Invalid</td>
<td>Feb. 24, 2010</td>
</tr>
<tr>
<td>2.3B</td>
<td>580 ACPI_SUPPORT_PROTOCOL Clarifications related to FADT and the DSĐT/FACS</td>
<td>Dec. 15, 2009</td>
</tr>
<tr>
<td>2.3B</td>
<td>578 ATA Passthrough updates / questions</td>
<td>Dec. 15, 2009</td>
</tr>
<tr>
<td>2.3B</td>
<td>577 Clarifications on the user identity protocol</td>
<td>Dec. 15, 2009</td>
</tr>
<tr>
<td>2.3B</td>
<td>576 Clarifications in the Routing Protocol</td>
<td>Dec. 15, 2009</td>
</tr>
<tr>
<td>2.3B</td>
<td>575 Machine hand-off/MP state modification</td>
<td>Feb. 24, 2010</td>
</tr>
<tr>
<td>2.3B</td>
<td>574 Add an &quot;OPTIONAL&quot; tag to a parameter in NewPackageList</td>
<td>Dec. 15, 2009</td>
</tr>
<tr>
<td>2.3B</td>
<td>573 EFI_DESCRIPTION_STRING and EFI_DESCRIPTION_BUNDLE adjustments</td>
<td>Dec. 15, 2009</td>
</tr>
<tr>
<td>2.3B</td>
<td>572 EFI_IFR_SECURITY should Be EFI_IFR_SECURITY_OP in Table 194</td>
<td>Dec. 15, 2009</td>
</tr>
<tr>
<td>2.3B</td>
<td>568 ATA_STATUS_BLOCK name errata</td>
<td>Dec. 15, 2009</td>
</tr>
<tr>
<td>2.3B</td>
<td>567 Various miscellaneous typos/updates</td>
<td>Feb. 24, 2010</td>
</tr>
<tr>
<td>2.3B</td>
<td>566 Minor update to HII-&gt;NewString function description</td>
<td>Dec. 15, 2009</td>
</tr>
<tr>
<td>2.3B</td>
<td>560 Correct erroneous example in ExtractConfig()</td>
<td>Dec. 15, 2009</td>
</tr>
<tr>
<td>2.3B</td>
<td>559 Extraneous “default” tag in EFI_IFR_SECURITY grammar</td>
<td>Dec. 15, 2009</td>
</tr>
<tr>
<td>2.3B</td>
<td>558 Clarify VLANConfig publication requirements</td>
<td>Dec. 15, 2009</td>
</tr>
<tr>
<td>2.3B</td>
<td>557 Corrected Image Execution Information omission &amp; ambiguity</td>
<td>Dec. 15, 2009</td>
</tr>
<tr>
<td>2.3B</td>
<td>556 Additional IPSec errata/Issues</td>
<td>Dec. 15, 2009</td>
</tr>
<tr>
<td>2.3B</td>
<td>549 Binary prefixChange</td>
<td>Dec. 15, 2009</td>
</tr>
<tr>
<td>2.3B</td>
<td>547 Clean-Up In HII Sections</td>
<td>Dec. 15, 2009</td>
</tr>
<tr>
<td>2.3B</td>
<td>546 Typo in GOP definition</td>
<td>Dec. 15, 2009</td>
</tr>
<tr>
<td>2.3B</td>
<td>545 Action parameter of the EFI_HII_CONFIG_ACCESS_PROTOCOL.CallBack()</td>
<td>Dec. 15, 2009</td>
</tr>
<tr>
<td>2.3B</td>
<td>542 Device Path DescriptionChanges</td>
<td>Dec. 15, 2009</td>
</tr>
<tr>
<td>2.3B</td>
<td>540 Register name usage</td>
<td>Dec. 15, 2009</td>
</tr>
<tr>
<td>2.3B</td>
<td>539 CHAP node fix for iSCSI</td>
<td>Dec. 15, 2009</td>
</tr>
<tr>
<td>2.3B</td>
<td>537 Add missing ACPI ADR Device Path Representation</td>
<td>Dec. 15, 2009</td>
</tr>
<tr>
<td>2.3B</td>
<td>536 IPSec errata</td>
<td>Dec. 15, 2009</td>
</tr>
<tr>
<td>2.3B</td>
<td>534 Size of Partition Entry restriction</td>
<td>Dec. 15, 2009</td>
</tr>
<tr>
<td>2.3B</td>
<td>533 GPT editorialCleanup</td>
<td>Dec. 15, 2009</td>
</tr>
<tr>
<td>2.3B</td>
<td>532 “LegacyBIOSBootable” GPT Attribute</td>
<td>Dec. 15, 2009</td>
</tr>
<tr>
<td>2.3B</td>
<td>531 Clarify HII Variable Storage</td>
<td>Dec. 15, 2009</td>
</tr>
<tr>
<td>2.3B</td>
<td>519 Add Console table (chapt 11) for EFI_SIMPLE_TEXST_INPUT_EX_PROTOCOL</td>
<td>Dec. 15, 2009</td>
</tr>
<tr>
<td>Revision</td>
<td>Mantis Number / Description</td>
<td>Date</td>
</tr>
<tr>
<td>----------</td>
<td>--------------------------------------------------------------------------------------------</td>
<td>------------</td>
</tr>
<tr>
<td>2.3B</td>
<td>518 Typos in the UEFI2.3 specification</td>
<td>Feb. 24, 2010</td>
</tr>
<tr>
<td>2.3B</td>
<td>515 Authenticated Variables Clarification</td>
<td>Feb. 24, 2010</td>
</tr>
<tr>
<td>2.3B</td>
<td>514 HII Configuration String Syntax Clarification</td>
<td>Feb. 24, 2010</td>
</tr>
<tr>
<td>2.3B</td>
<td>507 Clarify ACPI Protocol's position on Checksums</td>
<td>Dec. 15, 2009</td>
</tr>
<tr>
<td>2.3B</td>
<td>479 TPM guideline added to section 2.6.2</td>
<td>Dec. 15, 2009</td>
</tr>
<tr>
<td>2.3B</td>
<td>476 Text adjustment to ConfigAccess &amp; ConfigRouting</td>
<td>Dec. 15, 2009</td>
</tr>
<tr>
<td>2.3B</td>
<td>460 Section 2.6 language change</td>
<td>Dec. 15, 2009</td>
</tr>
<tr>
<td>2.3B</td>
<td>454 Dynamic support of media detection - network stack</td>
<td>Dec. 15, 2009</td>
</tr>
<tr>
<td>2.3B</td>
<td>431 UEFI 2.3 Feb Draft: Section 30.4</td>
<td>Feb. 24, 2010</td>
</tr>
<tr>
<td>2.3B</td>
<td>301 Errata to the Authentication Protocol</td>
<td>Dec. 15, 2009</td>
</tr>
<tr>
<td>2.3B</td>
<td>215 previously added to Device Driver (wrong), now BusDriver (correct)</td>
<td>Dec. 15, 2009</td>
</tr>
<tr>
<td>2.3A</td>
<td>522 Bugs in EFI_CERT_BLOCK_RSA_2048_SHA256, ISCSI device path, CHAP device path</td>
<td>Sept 15, 2009</td>
</tr>
<tr>
<td>2.3A</td>
<td>518 typos</td>
<td>Sept 15, 2009</td>
</tr>
<tr>
<td>2.3A</td>
<td>517 IP stack related protocol update</td>
<td>Sept 15, 2009</td>
</tr>
<tr>
<td>2.3A</td>
<td>516 User Identity Protocol Bugs</td>
<td>Sept 15, 2009</td>
</tr>
<tr>
<td>2.3A</td>
<td>513 add support for gateways in ipv4 &amp; ipv6 device path nodes</td>
<td>Sept 15, 2009</td>
</tr>
<tr>
<td>2.3A</td>
<td>506 TCP6/MTFTP6 StatusCode Definition</td>
<td>Sept 15, 2009</td>
</tr>
<tr>
<td>2.3A</td>
<td>505 TCP4/MTFTP4 StatusCode $</td>
<td>Sept 15, 2009</td>
</tr>
<tr>
<td>2.3A</td>
<td>490 Correction 28.2.5.6, Table 185. Information for Types of Storage</td>
<td>Sept 15, 2009</td>
</tr>
<tr>
<td>2.3A</td>
<td>478 Update to ALTCFG references</td>
<td>Sept 15, 2009</td>
</tr>
<tr>
<td>2.3A</td>
<td>477 Text adjustment to ConfigAccess/ConfigRouting</td>
<td>Sept 15, 2009</td>
</tr>
<tr>
<td>2.3</td>
<td>463 Update EFI_IP6_PROTOCOL.Neighbors() API</td>
<td>May 7, 2009</td>
</tr>
<tr>
<td>2.3</td>
<td>462 ExitBootServices timers deactivation</td>
<td>May 7, 2009</td>
</tr>
<tr>
<td>2.3</td>
<td>461 IP4 Mode Data definition update</td>
<td>May 7, 2009</td>
</tr>
<tr>
<td>2.3</td>
<td>460 Chapter 2.6 language update</td>
<td>May 7, 2009</td>
</tr>
<tr>
<td>2.3</td>
<td>457 Change KeyData.PackedValue to 0x40000200, page 63.</td>
<td>May 7, 2009</td>
</tr>
<tr>
<td>2.3</td>
<td>456 How to handle PXEBoot w/o NII Section 21.3</td>
<td>May 7, 2009</td>
</tr>
<tr>
<td>2.3</td>
<td>454 Dynamic support of media detection - network stack</td>
<td>May 7, 2009</td>
</tr>
<tr>
<td>2.3</td>
<td>453 Errata to support dynamic media detection - UNDI</td>
<td>May 7, 2009</td>
</tr>
<tr>
<td>2.3</td>
<td>452 Support to dynamically detect media errata - SNP</td>
<td>May 7, 2009</td>
</tr>
<tr>
<td>2.3</td>
<td>450 Missing opcode headers and formatting, section 28.3.8.3.x.</td>
<td>May 7, 2009</td>
</tr>
<tr>
<td>2.3</td>
<td>449 Add missing EFI_IFR_GET, EFI_IFR_SET and EFI_IFR_MAP to the syntax. Section 28.2.5.7.</td>
<td>May 7, 2009</td>
</tr>
<tr>
<td>2.3</td>
<td>448 Section 28.2.5.4 Questions, Syntax, Update question-option-tag; Add EFI_IFR_READ and EFI_IFR_WRITE in the question syntax.</td>
<td>May 7, 2009</td>
</tr>
<tr>
<td>2.3</td>
<td>447 Section 28.2.5.11.2 Moving Forms, Update line that starts with EFI_IFR_FORM to: EFI_IFR_FORM or EFI_IFR_FORM_MAP (and all references in EFI_IFR_REF)</td>
<td>May 7, 2009</td>
</tr>
<tr>
<td>Revision</td>
<td>Mantis Number / Description</td>
<td>Date</td>
</tr>
<tr>
<td>----------</td>
<td>----------------------------</td>
<td>--------------</td>
</tr>
<tr>
<td>2.3</td>
<td>446 Section 28.2.5.2 Forms, Syntax, Change 3rd line to: form := EFI_IFR_FORM form-tag-list</td>
<td>May 7, 2009</td>
</tr>
<tr>
<td></td>
<td>EFI_IFR_FORM_MAP form-tag-list</td>
<td></td>
</tr>
<tr>
<td>2.3</td>
<td>445 Table 194: EFI_IFR_FORM_MAP_OP, 2nd Column should be 0x5d (not 05xd)</td>
<td>May 7, 2009</td>
</tr>
<tr>
<td>2.3</td>
<td>444 Form Set Syntax: Section 28.2.5.1.1, section should be subheading, not heading level 5; Section 28.2.5.1, Syntax, line 3, text after := is not aligned with other text on line 2, 4</td>
<td>May 7, 2009</td>
</tr>
<tr>
<td>2.3</td>
<td>443 Section 28.3.8.3.38, EFI_IFR_MAP, Prototype, line 4, outdent 2 spaces.</td>
<td>May 7, 2009</td>
</tr>
<tr>
<td>2.3</td>
<td>442 Section 28.3.8.3.64, EFI_IFR_SET, Prototype, lines 3-8, indentBy 2 spaces</td>
<td>May 7, 2009</td>
</tr>
<tr>
<td>2.3</td>
<td>440 Change the defined type of EFI_STATUSES from INTN to UINTN</td>
<td>May 7, 2009</td>
</tr>
<tr>
<td>2.3</td>
<td>439 Incorrect definitions of UEFI_CONFIG_LANG and UEFI_CONFIG_LANG_2 in UEFI 2.3 Feb 18 draft</td>
<td>Feb 25, 2009</td>
</tr>
<tr>
<td>2.3</td>
<td>438 UEFI 2.3 Feb 13 Draft: Chapter 28 Formatting Issues</td>
<td>Feb 18, 2009</td>
</tr>
<tr>
<td>2.3</td>
<td>437 Errata to 2.3 draft material from UEFI Spec 2.3_Draft_Jan29</td>
<td>Feb 18, 2009</td>
</tr>
<tr>
<td>2.3</td>
<td>436 UEFI 2.3 split Figure 88 into 3 figures</td>
<td>Feb 12, 2009</td>
</tr>
<tr>
<td>2.3</td>
<td>435 Partition Signature Clarification</td>
<td>Feb 12, 2009</td>
</tr>
<tr>
<td>2.3</td>
<td>434 UEFI 2.3 Feb Draft: 28.3.8.3.58</td>
<td>Feb 12, 2009</td>
</tr>
<tr>
<td>2.3</td>
<td>432 UEFI 2.3 Feb Draft: Appendix M.</td>
<td>Feb 12, 2009</td>
</tr>
<tr>
<td>2.3</td>
<td>431 UEFI 2.3 Feb Draft: Section 30.4</td>
<td>Feb 12, 2009</td>
</tr>
<tr>
<td>2.3</td>
<td>418 Change Appendix O from &quot;UEFI ACPI Table&quot; to &quot;UEFI ACPI Data&quot;</td>
<td>Feb 18, 2009</td>
</tr>
<tr>
<td>2.3</td>
<td>413 Correct the definition of UEFI_CONFIG_LANG</td>
<td>Feb 18, 2009</td>
</tr>
<tr>
<td>2.3</td>
<td>410 UNDIBuffer usage</td>
<td>Feb 18, 2009</td>
</tr>
<tr>
<td>2.3</td>
<td>408 ARMBinding Corrections</td>
<td>Feb 12, 2009</td>
</tr>
<tr>
<td>2.3</td>
<td>406 Missing EFI System Table Revision In UEFI 2.3 Draft</td>
<td>Feb 12, 2009</td>
</tr>
<tr>
<td>2.3</td>
<td>395 New &quot;Non-removable MediaBootBehavior&quot; section</td>
<td>Feb 12, 2009</td>
</tr>
<tr>
<td>2.3</td>
<td>394 Omission in EFI_USB2_HC_PROTOCOL</td>
<td>Feb 12, 2009</td>
</tr>
<tr>
<td>2.3</td>
<td>388 Add HIICallback types (FORM_OPEN, FORM_CLOSE) when a form is opened or closed.</td>
<td>Feb 12, 2009</td>
</tr>
<tr>
<td>2.3</td>
<td>376 Add ARM processor Binding to UEFI</td>
<td>Jan 12, 2009</td>
</tr>
<tr>
<td>2.3</td>
<td>326 Add Firmware Management Protocol</td>
<td>Feb 12, 2009</td>
</tr>
<tr>
<td>2.2A</td>
<td>429 EFI_HASH_SERVICE_BINDING_PROTOCOL GUID define misses _GUID</td>
<td>Feb 12, 2009</td>
</tr>
<tr>
<td>2.2A</td>
<td>404 RemoveConstraint form EFI_TIME.YearComment</td>
<td>Feb 12, 2009</td>
</tr>
<tr>
<td>2.2A</td>
<td>400 FreePool() description error</td>
<td>Feb 12, 2009</td>
</tr>
<tr>
<td>2.2A</td>
<td>393 UEFI 2.1/2.2 Boot ManagerBehavior Clarification</td>
<td>Feb 12, 2009</td>
</tr>
<tr>
<td>2.2A</td>
<td>392 MBR errata in UEFI 2.2</td>
<td>Feb 12, 2009</td>
</tr>
<tr>
<td>2.2A</td>
<td>391 Polarity of INCONSISTENT_IF and NO_SUBMIT_IF IFR opcodes wrong</td>
<td>Feb 12, 2009</td>
</tr>
<tr>
<td>2.2A</td>
<td>390 UEFI 2.2 Miscellaneous HII-related errata</td>
<td>Feb 12, 2009</td>
</tr>
<tr>
<td>2.2A</td>
<td>389 UEFI 2.2 HII-Related Formatting Issues</td>
<td>Feb 12, 2009</td>
</tr>
<tr>
<td>2.2A</td>
<td>387 UEFI 2.1/UEFI 2.2A (ch. 12)</td>
<td>Feb 12, 2009</td>
</tr>
<tr>
<td>2.2A</td>
<td>384 Fix HII package description omission.</td>
<td>Feb 12, 2009</td>
</tr>
<tr>
<td>Revision</td>
<td>Mantis Number / Description</td>
<td>Date</td>
</tr>
<tr>
<td>----------</td>
<td>-----------------------------</td>
<td>------------</td>
</tr>
<tr>
<td>2.2A</td>
<td>379 UEFI 2.1/UEFI 2.2 HII-Related Errata</td>
<td>Feb. 12, 2009</td>
</tr>
<tr>
<td>2.2A</td>
<td>378 UEFI 2.1 &amp; UEFI 2.2 HIICallbackClarifications</td>
<td>Feb. 12, 2009</td>
</tr>
<tr>
<td>2.2A</td>
<td>377 MissingBLTBuffer figure.</td>
<td>Feb. 12, 2009</td>
</tr>
<tr>
<td>2.2A</td>
<td>375 Extra periods errata in UEFI 2.2</td>
<td>Feb. 12, 2009</td>
</tr>
<tr>
<td>2.2A</td>
<td>374 UEFI 2.1 &amp; UEFI 2.2A (10.7-10.10)</td>
<td>Feb. 12, 2009</td>
</tr>
<tr>
<td>2.2A</td>
<td>373 UEFI 2.2, Chs. 9.5 &amp; 9.6.2 &amp; 9.6.3 (Device Path) Errata</td>
<td>Feb. 12, 2009</td>
</tr>
<tr>
<td>2.2A</td>
<td>372 UEFI 2.2 remove “Draft for Review”</td>
<td>Feb. 12, 2009</td>
</tr>
<tr>
<td>2.2A</td>
<td>371 UEFI 2.1 &amp; UEFI 2.2 Typos (ch. 10)</td>
<td>Feb. 12, 2009</td>
</tr>
<tr>
<td>2.2A</td>
<td>370 EFI_SYSTEM_TABLE Errata (UEFI 2.1/UEFI 2.2)</td>
<td>Feb. 12, 2009</td>
</tr>
<tr>
<td>2.2A</td>
<td>368 EFI_FONT_DISPLAY_INFO.FontInfo description incorrect</td>
<td>Feb. 12, 2009</td>
</tr>
<tr>
<td>2.2A</td>
<td>366 UEFI 2.x: Erroneous references to EFI_BOOT_SERVICES_TABLE, EFI_RUNTIME_SERVICES_TABLE</td>
<td>Feb. 12, 2009</td>
</tr>
<tr>
<td>2.2A</td>
<td>364 UEFI 2.2 Typos &amp; Formatting Issues (ch. 9)</td>
<td>Feb. 12, 2009</td>
</tr>
<tr>
<td>2.2A</td>
<td>362 UEFI 2.2 Typos (Next)</td>
<td>Feb. 12, 2009</td>
</tr>
<tr>
<td>2.2A</td>
<td>361 UEFI 2.2 Typos &amp; Formatting Issues</td>
<td>Feb. 12, 2009</td>
</tr>
<tr>
<td>2.2A</td>
<td>359 TPL Table</td>
<td>Feb. 12, 2009</td>
</tr>
<tr>
<td>2.2A</td>
<td>358 Missing signature for UEFI 2.2.</td>
<td>Feb. 12, 2009</td>
</tr>
<tr>
<td>2.2</td>
<td>398 Update to M348 to fix small typo</td>
<td>Jan. 11, 2009</td>
</tr>
<tr>
<td>2.2</td>
<td>397 PCICopyMem() misspelling</td>
<td>Jan. 11, 2009</td>
</tr>
<tr>
<td>2.2</td>
<td>394 Omission in EFI_USB2_HC_PROTOCOL</td>
<td>Jan. 11, 2009</td>
</tr>
<tr>
<td>2.2</td>
<td>357 Clarify EFI_IFR_DISABLE_IFBehavior with regard to dynamic values</td>
<td>Jan. 11, 2009</td>
</tr>
<tr>
<td>2.2</td>
<td>351 Fix an unaligned field in a device path</td>
<td>Jan. 11, 2009</td>
</tr>
<tr>
<td>2.2</td>
<td>350 EFI_HII_STRING_PROTOCOL Typos</td>
<td>Jan. 11, 2009</td>
</tr>
<tr>
<td>2.2</td>
<td>348 EFI_IFR_RESET_BUTTON is incorrectly listed as a question</td>
<td>Jan. 11, 2009</td>
</tr>
<tr>
<td>2.2</td>
<td>347 Replace first paragraph of the “Description” section for the ExitBootServices()</td>
<td>Sept. 25, 2008</td>
</tr>
<tr>
<td>2.2</td>
<td>346 Nest, Sections 10.11 &amp; 10.12 Under 10.10</td>
<td>Sept. 25, 2008</td>
</tr>
<tr>
<td>2.2</td>
<td>344 Correct missing statusCodes returned section for Form() in EFI_USER_CREDENTIAL_PROTOCOL.</td>
<td>Sept. 25, 2008</td>
</tr>
<tr>
<td>2.2</td>
<td>343 Correct missing parameter for User() function in EFI_USER_CREDENTIAL_PROTOCOL</td>
<td>Sept. 25, 2008</td>
</tr>
<tr>
<td>2.2</td>
<td>340 UEFI 2.2 Editorial / Formatting Issues</td>
<td>Sept. 25, 2008</td>
</tr>
<tr>
<td>2.2</td>
<td>339 Update missing TPL restrictions</td>
<td>Sept. 25, 2008</td>
</tr>
<tr>
<td>2.2</td>
<td>337 Replace the EFI_CRYPT_HANDLE reference (in the IPSec API) with a self-contained, independent definition.</td>
<td>Sept. 25, 2008</td>
</tr>
<tr>
<td>2.2</td>
<td>335 User Authentication errata</td>
<td>Sept. 25, 2008</td>
</tr>
<tr>
<td>2.2</td>
<td>334 Standardized &quot;Unicode&quot; References</td>
<td>Jan. 11, 2009</td>
</tr>
<tr>
<td>2.2</td>
<td>333 Correct the incorrect ';' at the end of EFI_GUID #defines</td>
<td>Sept. 25, 2008</td>
</tr>
<tr>
<td>2.2</td>
<td>332 Correct SendForm description Type, PackageGuid and FormsetGuid parameters</td>
<td>Sept. 25, 2008</td>
</tr>
<tr>
<td>2.2</td>
<td>331 Definition for EFI_BROWSER_ACTION and the related #defines were not present--Insert.</td>
<td>Sept. 25, 2008</td>
</tr>
<tr>
<td>Revision</td>
<td>Mantis Number / Description</td>
<td>Date</td>
</tr>
<tr>
<td>----------</td>
<td>-----------------------------</td>
<td>------------</td>
</tr>
<tr>
<td>2.2</td>
<td>330 EFI_IFR_REF:ChangeCross reference to a question</td>
<td>Sept. 25, 2008</td>
</tr>
<tr>
<td>2.2</td>
<td>327 Clarify the support in DHCP4 protocol for &quot;Inform&quot; (DHCPINFORM) messages.</td>
<td>Sept. 25, 2008</td>
</tr>
<tr>
<td>2.2</td>
<td>325 MinorCorrection 28.3.8.3.20</td>
<td>July 25, 2008</td>
</tr>
<tr>
<td>2.2</td>
<td>324 ATA Pass-Thru ECR Update</td>
<td>July 25, 2008</td>
</tr>
<tr>
<td>2.2</td>
<td>323 VLAN modification Because of IPV6</td>
<td>July 25, 2008</td>
</tr>
<tr>
<td>2.2</td>
<td>322 Chapter 2 updates for IP6 net stack</td>
<td>July 25, 2008</td>
</tr>
<tr>
<td>2.2</td>
<td>321 Enable PCIe 2.0 and Beyond support in the UEFI error records</td>
<td>July 25, 2008</td>
</tr>
<tr>
<td>2.2</td>
<td>320 Clarification for WIN_CERTIFICATE types &amp; relationship with signature database types</td>
<td>July 25, 2008</td>
</tr>
<tr>
<td>2.2</td>
<td>319 UEFI IPSec protocol</td>
<td>July 25, 2008</td>
</tr>
<tr>
<td>2.2</td>
<td>315 EFI TC6 Protocol</td>
<td>July 25, 2008</td>
</tr>
<tr>
<td>2.2</td>
<td>314 EFI MTFTP6 Protocol</td>
<td>July 25, 2008</td>
</tr>
<tr>
<td>2.2</td>
<td>313 EFI IPv6 Configuration Protocol</td>
<td>July 25, 2008</td>
</tr>
<tr>
<td>2.2</td>
<td>312 EFI IPv6 Protocol</td>
<td>July 25, 2008</td>
</tr>
<tr>
<td>2.2</td>
<td>311 EFI DHCPv6 Protocol</td>
<td>July 25, 2008</td>
</tr>
<tr>
<td>2.2</td>
<td>310 EFI UDPv6 Protocol</td>
<td>July 25, 2008</td>
</tr>
<tr>
<td>2.2</td>
<td>309 IPv6 Address display format Clarification</td>
<td>July 25, 2008</td>
</tr>
<tr>
<td>2.2</td>
<td>306 Some errata to the animation support</td>
<td>July 25, 2008</td>
</tr>
<tr>
<td>2.2</td>
<td>304 Errata to UpdateCapsule()</td>
<td>July 25, 2008</td>
</tr>
<tr>
<td>2.2</td>
<td>303 Add ability to have a Capsule that initiates a reset &amp; doesn’t return to the Caller</td>
<td>July 25, 2008</td>
</tr>
<tr>
<td>2.2</td>
<td>301 Errata to the Authentication Protocol</td>
<td>July 25, 2008</td>
</tr>
<tr>
<td>2.2</td>
<td>300 MTFTP errata</td>
<td>July 25, 2008</td>
</tr>
<tr>
<td>2.2</td>
<td>299 PIWG Firmware File/Firmware Volume Typo Errata</td>
<td>July 25, 2008</td>
</tr>
<tr>
<td>2.2</td>
<td>294 LocateDevicePath with multi-instance device path</td>
<td>July 25, 2008</td>
</tr>
<tr>
<td>2.2</td>
<td>291 HII Errata / Update</td>
<td>July 25, 2008</td>
</tr>
<tr>
<td>2.2</td>
<td>288 Additional wording fixes for GPT Entry Attribute Bit 1</td>
<td>July 25, 2008</td>
</tr>
<tr>
<td>2.2</td>
<td>282 Updated Requirements Section For ATA Pass Through (M242)</td>
<td>July 25, 2008</td>
</tr>
<tr>
<td>2.2</td>
<td>279 Firmware/OS Trusted Key Exchange and Image Validation</td>
<td>July 25, 2008</td>
</tr>
<tr>
<td>2.2</td>
<td>237 UEFI User Identification Proposal (from USST)</td>
<td>July 25, 2008</td>
</tr>
<tr>
<td>2.2</td>
<td>215 new Start() RemainingDevicePath Syntax</td>
<td>July 25, 2008</td>
</tr>
<tr>
<td>2.2</td>
<td>212 UEFI HII Standards Mapping</td>
<td>July 25, 2008</td>
</tr>
<tr>
<td>2.2</td>
<td>211 UEFI Setup Question / Form Access Update</td>
<td>July 25, 2008</td>
</tr>
<tr>
<td>2.2</td>
<td>210 UEFI HII Animation addition</td>
<td>July 25, 2008</td>
</tr>
<tr>
<td>2.2</td>
<td>202 EAP Management</td>
<td>July 25, 2008</td>
</tr>
<tr>
<td>2.2</td>
<td>201EAP</td>
<td>July 25, 2008</td>
</tr>
<tr>
<td>2.2</td>
<td>200 VLAN</td>
<td>July 25, 2008</td>
</tr>
<tr>
<td>2.2</td>
<td>199 FTP API</td>
<td>July 25, 2008</td>
</tr>
<tr>
<td>2.2</td>
<td>198 GUID Partition Entry Attributes Clarification and Definition</td>
<td>July 25, 2008</td>
</tr>
<tr>
<td>Revision</td>
<td>Mantis Number / Description</td>
<td>Date</td>
</tr>
<tr>
<td>----------</td>
<td>-----------------------------</td>
<td>---------------</td>
</tr>
<tr>
<td>2.2</td>
<td>169 EFI Driver Health Protocol</td>
<td>July 25, 2008</td>
</tr>
<tr>
<td>2.2</td>
<td>157 Floating-Point ABI Changes For X86, X64 &amp; Itanium</td>
<td>July 25, 2008</td>
</tr>
<tr>
<td>2.1C</td>
<td>Re-format Revision History from Bulleted lists to one row per Mantis ticket/EngineeringChange Request</td>
<td>June 5, 2008</td>
</tr>
<tr>
<td>2.1C</td>
<td>60 iSCSI Device Path Update</td>
<td>June 5, 2008</td>
</tr>
<tr>
<td>2.1C</td>
<td>59 Add returnCode to Diagnostics Protocol</td>
<td>June 5, 2008</td>
</tr>
<tr>
<td>2.1C</td>
<td>58 Language update for EfiReservedMemory type usage</td>
<td>June 5, 2008</td>
</tr>
<tr>
<td>2.1C</td>
<td>56 Clarification on ResetSystem</td>
<td>June 5, 2008</td>
</tr>
<tr>
<td>2.1C</td>
<td>55 Clarification on UpdateCapsule</td>
<td>June 5, 2008</td>
</tr>
<tr>
<td>2.1C</td>
<td>54 ACPI Table Protocol GUID Update</td>
<td>June 5, 2008</td>
</tr>
<tr>
<td>2.1C</td>
<td>52 New GUID for Driver Diagnostics and DriverConfiguration Protocols with new GUID</td>
<td>June 5, 2008</td>
</tr>
<tr>
<td>2.1C</td>
<td>283 Minor update to Clarify a typedef/returnCode in HII</td>
<td>June 5, 2008</td>
</tr>
<tr>
<td>2.1C</td>
<td>281 Runtime memory allocation</td>
<td>June 5, 2008</td>
</tr>
<tr>
<td>2.1C</td>
<td>280 Some minor errata to keyboard related topics</td>
<td>June 5, 2008</td>
</tr>
<tr>
<td>2.1C</td>
<td>278 Change references to EFI_SIMPLE_INPUT_PROTOCOL into EFI_SIMPLE_TEXT_INPUT_PROTOCOL</td>
<td>June 5, 2008</td>
</tr>
<tr>
<td>2.1C</td>
<td>266 PKCS11.5 structure does not Correctly specify the portion of the Cited RFC that pertains to the Certificate struct/algorithm</td>
<td>June 5, 2008</td>
</tr>
<tr>
<td>2.1C</td>
<td>249 Latest update to UCST Errata list</td>
<td>June 5, 2008</td>
</tr>
<tr>
<td>2.1C</td>
<td>248 Correction to text in Chapter 8.2 of UEFI 2.1B</td>
<td>June 5, 2008</td>
</tr>
<tr>
<td>2.1C</td>
<td>246 New returnCode</td>
<td>June 5, 2008</td>
</tr>
<tr>
<td>2.1C</td>
<td>245 Remove extraneous text in Chapter 29</td>
<td>June 5, 2008</td>
</tr>
<tr>
<td>2.1C</td>
<td>244 Replace references to EFI_FIRMWARE_VOLUME_INFO_PPI with EFI_PEI_FIRMWARE_VOLUME_INFO_PPI</td>
<td>June 5, 2008</td>
</tr>
<tr>
<td>2.1C</td>
<td>221 ImageBlock Structure name typos in 27.3.7.2</td>
<td>June 5, 2008</td>
</tr>
<tr>
<td>2.1C</td>
<td>220 Replace references to RFC 3066 to RFC 4646</td>
<td>June 5, 2008</td>
</tr>
<tr>
<td>2.1C</td>
<td>219 IA-32 and x64 stack need to be 16-byte aligned</td>
<td>June 5, 2008</td>
</tr>
<tr>
<td>2.1C</td>
<td>218 SATA update to section 9.3.5.6</td>
<td>June 5, 2008</td>
</tr>
<tr>
<td>2.1C</td>
<td>217 EFI_PLATFORM_TO_DRIVER_CONFIGURATION_PROTOCOL.Query() Update</td>
<td>June 5, 2008</td>
</tr>
<tr>
<td>2.1C</td>
<td>216 UEFI 2.1 text Corrections</td>
<td>June 5, 2008</td>
</tr>
<tr>
<td>2.1C</td>
<td>214 Device_IO + typos</td>
<td>June 5, 2008</td>
</tr>
<tr>
<td>2.1C</td>
<td>213 UEFI HII Errata</td>
<td>June 5, 2008</td>
</tr>
<tr>
<td>2.1C</td>
<td>209 ESP number/location Clarifications</td>
<td>June 5, 2008</td>
</tr>
<tr>
<td>2.1C</td>
<td>208 Driver Protocol Names and GUIDs</td>
<td>June 5, 2008</td>
</tr>
<tr>
<td>2.1C</td>
<td>207 Updated Wording for the File Path</td>
<td>June 5, 2008</td>
</tr>
<tr>
<td>2.1C</td>
<td>206 Clarify return values for extended scsi passthru protocol</td>
<td>June 5, 2008</td>
</tr>
<tr>
<td>2.1C</td>
<td>203 Platform Error Record - x64 register state errata</td>
<td>June 5, 2008</td>
</tr>
<tr>
<td>2.1C</td>
<td>193 Loaded Image device paths for EFI Drivers loaded from PCI Option ROMs</td>
<td>June 5, 2008</td>
</tr>
<tr>
<td>Revision</td>
<td>Mantis Number / Description</td>
<td>Date</td>
</tr>
<tr>
<td>----------</td>
<td>----------------------------</td>
<td>-----------------</td>
</tr>
<tr>
<td>2.1B</td>
<td>51 Long physicalBlocks updates</td>
<td>December 11, 2007</td>
</tr>
<tr>
<td>2.1B</td>
<td>205 Change LoadImage() parameter name from FilePath to DevicePath; endsConfusion with EFI_LOADED_IMAGE_PROTOCOL</td>
<td>December 11, 2007</td>
</tr>
<tr>
<td>2.1B</td>
<td>197 EFI Loaded Image Device Path Protocol</td>
<td>December 11, 2007</td>
</tr>
<tr>
<td>2.1B</td>
<td>190 Extensive errata form UCST including OPCodesChanges ro resolveConflicts.</td>
<td>December 11, 2007</td>
</tr>
<tr>
<td>2.1B</td>
<td>187 Clarify input protocols.</td>
<td>December 11, 2007</td>
</tr>
<tr>
<td>2.1B</td>
<td>186 Change PCIR struct to match PCI FW Spec 3.0</td>
<td>December 11, 2007</td>
</tr>
<tr>
<td>2.1B</td>
<td>185 Change EFI term to UEFI forConsistency</td>
<td>December 11, 2007</td>
</tr>
<tr>
<td>2.1B</td>
<td>184 SNIA/DDF Wording Update</td>
<td>December 11, 2007</td>
</tr>
<tr>
<td>2.1B</td>
<td>182 Clarify EFI_MTFTP4_TOKEN</td>
<td>December 11, 2007</td>
</tr>
<tr>
<td>2.1B</td>
<td>181 Correct MNP GUIDCollision</td>
<td>December 11, 2007</td>
</tr>
<tr>
<td>2.1B</td>
<td>177 remove ending paragraph (editing text) in section 9.6</td>
<td>December 11, 2007</td>
</tr>
<tr>
<td>2.1B</td>
<td>175 Update to SendForm API</td>
<td>December 11, 2007</td>
</tr>
<tr>
<td>2.1B</td>
<td>174 Error record addition for dma remapping units</td>
<td>December 11, 2007</td>
</tr>
<tr>
<td>2.1B</td>
<td>173 Minor Changes to the description of two of the fields in the Common Platform Error Record, in Appendix N</td>
<td>December 11, 2007</td>
</tr>
<tr>
<td>2.1B</td>
<td>172 Typo for ResetSystem()</td>
<td>December 11, 2007</td>
</tr>
<tr>
<td>2.1B</td>
<td>170 (Addition of) Driver Family Override Protocol</td>
<td>December 11, 2007</td>
</tr>
<tr>
<td>2.1B</td>
<td>168 Remove LOAD_OPTION_GRAPHICS</td>
<td>December 11, 2007</td>
</tr>
<tr>
<td>2.1B</td>
<td>165 Fix EFI_GRAPHICS_OUTPUT_PIXEL</td>
<td>December 11, 2007</td>
</tr>
<tr>
<td>2.1B</td>
<td>164 Update to USB2_HC_PROTOCOL Table</td>
<td>December 11, 2007</td>
</tr>
<tr>
<td>2.1B</td>
<td>162 UEFI PIWG Device Path Errata</td>
<td>December 11, 2007</td>
</tr>
<tr>
<td>2.1B</td>
<td>160 Clean up references to PCIR</td>
<td>December 11, 2007</td>
</tr>
<tr>
<td>2.1B</td>
<td>158 Errata to the UEFI 2.1 Configuration sections</td>
<td>December 11, 2007</td>
</tr>
<tr>
<td>2.1B</td>
<td>156 SendForm API Errata</td>
<td>December 11, 2007</td>
</tr>
<tr>
<td>2.1B</td>
<td>159 Adjust some of the #define names in the Simple Text Input Ex protocol</td>
<td>December 11, 2007</td>
</tr>
<tr>
<td>2.1A</td>
<td>UEFI 2.1 incorporating Errata through 4-27-07</td>
<td>April 27, 2007</td>
</tr>
<tr>
<td>2.1</td>
<td>Second release</td>
<td>January 23, 2007</td>
</tr>
<tr>
<td>2.0</td>
<td>First release of specification.</td>
<td>January 31, 2006</td>
</tr>
</tbody>
</table>
# Table of Contents

## Acknowledgments ........................................................................................................ ii
## Revision History ........................................................................................................ iii
## Table of Contents ....................................................................................................... xxviii
## List of Tables ................................................................................................................ lxiv
## List of Figures .............................................................................................................. lxxii

1 Introduction .................................................................................................................. 1
   1.1 UEFI Driver Model Extensions ........................................................................... 1
   1.2 Organization ......................................................................................................... 2
   1.3 Goals ..................................................................................................................... 4
   1.4 Target Audience ................................................................................................... 6
   1.5 UEFI Design Overview ....................................................................................... 7
   1.6 UEFI Driver Model .............................................................................................. 8
      1.6.1 UEFI Driver Model Goals ........................................................................... 8
      1.6.2 Legacy Option ROM Issues ......................................................................... 9
   1.7 Migration Requirements ...................................................................................... 9
      1.7.1 Legacy Operating System Support ............................................................... 9
      1.7.2 Supporting the UEFI Specification on a Legacy Platform ......................... 10
   1.8 Conventions Used in this Document ................................................................... 10
      1.8.1 Data Structure Descriptions ...................................................................... 10
      1.8.2 Protocol Descriptions ................................................................................ 10
      1.8.3 Procedure Descriptions ............................................................................. 10
      1.8.4 Instruction Descriptions ............................................................................ 11
      1.8.5 Pseudo-Code Conventions ....................................................................... 11
      1.8.6 Typographic Conventions ........................................................................ 11
      1.8.7 Number formats ......................................................................................... 12
      1.8.8 Binary prefixes ......................................................................................... 13
      1.8.9 Revision Numbers .................................................................................... 13

2 Overview ....................................................................................................................... 14
   2.1 Boot Manager ..................................................................................................... 14
      2.1.1 UEFI Images ............................................................................................... 15
      2.1.2 UEFI Applications ...................................................................................... 16
      2.1.3 UEFI OS Loaders ....................................................................................... 16
      2.1.4 UEFI Drivers ............................................................................................ 17
   2.2 Firmware Core .................................................................................................... 17
      2.2.1 UEFI Services ............................................................................................ 17
      2.2.2 Runtime Services ....................................................................................... 18
   2.3 Calling Conventions ........................................................................................... 19
      2.3.1 Data Types ............................................................................................... 20
      2.3.2 IA-32 Platforms ....................................................................................... 21
      2.3.3 Intel® Itanium®-Based Platforms ............................................................... 24
# UEFI Specification, Version 2.9

**Table of Contents**

## 2 Protocols

2.1 Required Elements ........................................................................................................... 45
2.2 Platform-Specific Elements ............................................................................................... 46
2.3 Driver-Specific Elements ..................................................................................................... 46
2.4 Extensions to this Specification published elsewhere .......................................................... 47

## 2.5 UEFI Driver Model

2.5.1 Legacy Option ROM Issues .............................................................................................. 50
2.5.2 Driver Initialization ............................................................................................................. 52
2.5.3 Host Bus Controllers .......................................................................................................... 54
2.5.4 Device Drivers .................................................................................................................... 55
2.5.5 Bus Drivers ......................................................................................................................... 57
2.5.6 Platform Components ........................................................................................................ 58
2.5.7 Hot-Plug Events ............................................................................................................... 60
2.5.8 EFI Services Binding ......................................................................................................... 60

## 2.6 Requirements

2.6.1 Required Elements ............................................................................................................. 62
2.6.2 Platform-Specific Elements ............................................................................................... 63
2.6.3 Driver-Specific Elements ..................................................................................................... 66
2.6.4 Extensions to this Specification published elsewhere .......................................................... 68

## 3 Boot Manager

3.1 Firmware Boot Manager ...................................................................................................... 70
3.1.1 Boot Manager Programming ............................................................................................. 71
3.1.2 Load Option Processing ..................................................................................................... 72
3.1.3 Load Options ....................................................................................................................... 73
3.1.4 Boot Manager Capabilities ............................................................................................... 75
3.1.5 Launching Boot## Applications .................................................................................... 75
3.1.6 Launching Boot### Load Options Using Hot Keys ............................................................. 76
3.1.7 Required System Preparation Applications ....................................................................... 78
3.2 Boot Manager Policy Protocol ............................................................................................ 78
3.2.1 EFI_BOOT_MANAGER_POLICY_PROTOCOL ......................................................... 78
3.2.2 EFI_BOOT_MANAGER_POLICY_PROTOCOL.ConnectDevicePath() ....................... 79
3.2.3 EFI_BOOT_MANAGER_POLICY_PROTOCOL.ConnectDeviceClass() ...................... 80
3.3 Globally Defined Variables ................................................................................................. 81
3.4 Boot Option Recovery ......................................................................................................... 87
3.4.1 OS-Defined Boot Option Recovery .................................................................................. 87
3.4.2 Platform-Defined Boot Option Recovery ........................................................................ 88
3.4.3 Boot Option Variables Default Boot Behavior ................................................................. 88
3.5 Boot Mechanisms ................................................................................................................ 88
3.5.1 Boot via the Simple File Protocol .................................................................................... 88
3.5.2 Boot via the Load File Protocol ....................................................................................... 89

## 4 EFI System Table

4.1 UEFI Image Entry Point ...................................................................................................... 91
4.2 EFI Table Header ................................................................................................................ 92
7 Services — Boot Services .......................... 143

7.1 Event, Timer, and Task Priority Services .......................................................... 144
  EFI_BOOT_SERVICES.CreateEvent() .............................................................. 148
  EFI_BOOT_SERVICES.CreateEventEx() ......................................................... 152
  EFI_BOOT_SERVICES.CloseEvent() ............................................................... 156
  EFI_BOOT_SERVICES.SignalEvent() ............................................................. 157
  EFI_BOOT_SERVICES.WaitForEvent() ......................................................... 158
  EFI_BOOT_SERVICES.CheckEvent() ............................................................. 159
  EFI_BOOT_SERVICES.SetTimer() ................................................................. 160
  EFI_BOOT_SERVICES.RaiseTPL() ................................................................. 161
  EFI_BOOT_SERVICES.RestoreTPL() ............................................................. 162

7.2 Memory Allocation Services ........................................................................ 163
  EFI_BOOT_SERVICES.AllocatePages() .......................................................... 165
  EFI_BOOT_SERVICES.FreePages() ............................................................... 168
  EFI_BOOT_SERVICES.GetMemoryMap() ....................................................... 169
  EFI_BOOT_SERVICES.AllocatePool() ........................................................... 173
  EFI_BOOT_SERVICES.FreePool() ................................................................. 174

7.3 Protocol Handler Services .............................................................................. 175
  EFI_BOOT_SERVICES.InstallProtocolInterface() .......................................... 179
  EFI_BOOT_SERVICES.UninstallProtocolInterface() ...................................... 182
  EFI_BOOT_SERVICES.ReinstallProtocolInterface() .................................... 183
  EFI_BOOT_SERVICES.RegisterProtocolNotify() .......................................... 185
  EFI_BOOT_SERVICES.LocateHandle() .......................................................... 186
  EFI_BOOT_SERVICES.HandleProtocol() ...................................................... 188
  EFI_BOOT_SERVICES.LocateDevicePath() .................................................. 190
  EFI_BOOT_SERVICES.OpenProtocol() .......................................................... 191
  EFI_BOOT_SERVICES.CloseProtocol() .......................................................... 199
  EFI_BOOT_SERVICES.OpenProtocolInformation() ....................................... 201
  EFI_BOOT_SERVICES.ConnectController() .................................................. 202
  EFI_BOOT_SERVICES.DisconnectController() ............................................. 207
  EFI_BOOT_SERVICES.ProtocolsPerHandle() ................................................. 209
  EFI_BOOT_SERVICES.LocateHandleBuffer() .............................................. 210
  EFI_BOOT_SERVICES.LocateProtocol() ....................................................... 213
  EFI_BOOT_SERVICES.InstallMultipleProtocolInterfaces() ......................... 214
  EFI_BOOT_SERVICES.UninstallMultipleProtocolInterfaces() ..................... 215

7.4 Image Services ............................................................................................ 216
  EFI_BOOT_SERVICES.LoadImage() ............................................................... 217
  EFI_BOOT_SERVICES.StartImage() ............................................................... 220
  EFI_BOOT_SERVICES.UnloadImage() ........................................................... 221
  EFI_IMAGE_ENTRY_POINT ........................................................................... 222
  EFI_BOOT_SERVICES.Exit() ........................................................................ 223
  EFI_BOOT_SERVICES.ExitBootServices() .................................................... 225

7.5 Miscellaneous Boot Services .......................................................................... 226
  EFI_BOOT_SERVICES.SetWatchdogTimer() .................................................. 227
  EFI_BOOT_SERVICES.Stall() ........................................................................ 228
  EFI_BOOT_SERVICES.CopyMem() ............................................................... 228
### 8 Services — Runtime Services

**8.1 Runtime Services Rules and Restrictions** ................................................. 233

**8.2 Variable Services** ...................................................................................... 236

- GetVariable() ........................................................................................................ 236
- GetNextVariableName() ...................................................................................... 239
- SetVariable() ........................................................................................................ 241
- QueryVariableInfo() ........................................................................................... 247

**8.2.1 Using the EFI_VARIABLE_AUTHENTICATION_3 descriptor** ................... 249

**8.2.2 Using the EFI_VARIABLE_AUTHENTICATION_2 descriptor** .................... 252

**8.2.3 Using the EFI_VARIABLE_AUTHENTICATION descriptor** ......................... 255

**8.2.4 Hardware Error Record Persistence** ....................................................... 256

**8.3 Time Services** .......................................................................................... 257

- GetTime() .............................................................................................................. 257
- SetTime() .............................................................................................................. 261
- GetWakeupTime() ................................................................................................. 262
- SetWakeupTime() ................................................................................................. 263

**8.4 Virtual Memory Services** ........................................................................... 264

- SetVirtualAddressMap() ..................................................................................... 265
- ConvertPointer() .................................................................................................. 266

**8.5 Miscellaneous Runtime Services** ............................................................... 267

**8.5.1 Reset System** .......................................................................................... 268

- ResetSystem() ..................................................................................................... 268

**8.5.2 Get Next High Monotonic Count** .............................................................. 269

- GetNextHighMonotonicCount() ........................................................................... 269

**8.5.3 Update Capsule** ....................................................................................... 271

- UpdateCapsule() ................................................................................................ 271
- EFI_MEMORY_RANGE_CAPSULE_GUID .................................................................. 278
- QueryCapsuleCapabilities() ................................................................................ 279

**8.5.4 Exchanging information between the OS and Firmware** ......................... 280

**8.5.5 Delivery of Capsules via file on Mass Storage device** ......................... 282

**8.5.6 UEFI variable reporting on the Success or any Errors encountered in processing of capsules after restart** ......................................................... 283

### 9 Protocols — EFI Loaded Image

**9.1 EFI Loaded Image Protocol** ................................................................. 288

- EFI_LOADED_IMAGE_PROTOCOL ...................................................................... 288
- EFI_LOADED_IMAGE_PROTOCOL.Unload() ...................................................... 289

**9.2 EFI Loaded Image Device Path Protocol** .............................................. 290

- EFI_LOADED_IMAGE_DEVICE_PATH_PROTOCOL ............................................. 290
10 Protocols — Device Path Protocol ................................................................. 291

10.1 Device Path Overview ........................................................................... 291
10.2 EFI Device Path Protocol ..................................................................... 291
    EFI_DEVICE_PATH_PROTOCOL ...................................................................... 291
10.3 Device Path Nodes ................................................................................ 292
    10.3.1 Generic Device Path Structures ..................................................... 293
    10.3.2 Hardware Device Path ..................................................................... 294
    10.3.3 ACPI Device Path ........................................................................... 296
    10.3.4 Messaging Device Path ................................................................. 299
    10.3.5 Media Device Path .......................................................................... 325
    10.3.6 BIOS Boot Specification Device Path ............................................. 330
10.4 Device Path Generation Rules ................................................................. 331
    10.4.1 Housekeeping Rules ...................................................................... 331
    10.4.2 Rules with ACPI _HID and _UID .................................................... 332
    10.4.3 Rules with ACPI _ADR .................................................................... 332
    10.4.4 Hardware vs. Messaging Device Path Rules .................................. 333
    10.4.5 Media Device Path Rules ............................................................... 333
    10.4.6 Other Rules ................................................................................... 334
10.5 Device Path Utilities Protocol ................................................................. 334
    EFI_DEVICE_PATH_UTILITIES_PROTOCOL ........................................... 334
    EFI_DEVICE_PATH_UTILITIES_PROTOCOL.GetDevicePathSize() ................. 335
    EFI_DEVICE_PATH_UTILITIES_PROTOCOL.DuplicateDevicePath() ............. 335
    EFI_DEVICE_PATH_UTILITIES_PROTOCOL.AppendDevicePath() ................. 336
    EFI_DEVICE_PATH_UTILITIES_PROTOCOL.AppendDeviceNode() ............... 337
    EFI_DEVICE_PATH_UTILITIES_PROTOCOL.AppendDevicePathInstance() ...... 337
    EFI_DEVICE_PATH_UTILITIES_PROTOCOL.GetNextDevicePathInstance() ... 338
    EFI_DEVICE_PATH_UTILITIES_PROTOCOL.CreateDeviceNode() ................ 339
    EFI_DEVICE_PATH_UTILITIES_PROTOCOL.IsDevicePathMultiInstance() ...... 340
10.6 EFI Device Path Display Format Overview ............................................ 340
    10.6.1 Design Discussion ....................................................................... 341
    10.6.2 Device Path to Text Protocol ....................................................... 356
        EFI_DEVICE_PATH_TO_TEXT_PROTOCOL .......................................... 356
        EFI_DEVICE_PATH_TO_TEXT_PROTOCOL.ConvertDeviceNodeToText() .... 357
        EFI_DEVICE_PATH_TO_TEXT_PROTOCOL.ConvertDevicePathToText() ...... 358
    10.6.3 Device Path from Text Protocol .................................................... 358
        EFI_DEVICE_PATH_FROM_TEXT_PROTOCOL ...................................... 358
        EFI_DEVICE_PATH_FROM_TEXT_PROTOCOL.ConvertTextToDeviceNode() .... 359
        EFI_DEVICE_PATH_FROM_TEXT_PROTOCOL.ConvertTextToDevicePath() ... 360

11 Protocols — UEFI Driver Model ............................................................... 361

11.1 EFI Driver Binding Protocol ................................................................. 361
    EFI_DRIVER_BINDING_PROTOCOL ......................................................... 361
    EFI_DRIVER_BINDING_PROTOCOL_SUPPORTED() .................................... 363
    EFI_DRIVER_BINDING_PROTOCOL_START() ........................................... 368
    EFI_DRIVER_BINDING_PROTOCOL_STOP() ............................................. 377
11.2 EFI Platform Driver Override Protocol .................................................. 381
<table>
<thead>
<tr>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>11.3 EFI Bus Specific Driver Override Protocol</td>
<td>386</td>
</tr>
<tr>
<td>EFI_BUS_SPECIFIC_DRIVER.Override_PROTOCOL.GetDriver()</td>
<td>387</td>
</tr>
<tr>
<td>11.4 EFI Driver Diagnostics Protocol</td>
<td>388</td>
</tr>
<tr>
<td>EFI_DRIVER_DIAGNOSTICS2_PROTOCOL.RunDiagnostics()</td>
<td>389</td>
</tr>
<tr>
<td>11.5 EFI Component Name Protocol</td>
<td>392</td>
</tr>
<tr>
<td>EFI_COMPONENT_NAME2_PROTOCOL.GetDriverName()</td>
<td>393</td>
</tr>
<tr>
<td>EFI_COMPONENT_NAME2_PROTOCOL.GetControllerName()</td>
<td>394</td>
</tr>
<tr>
<td>11.6 EFI Service Binding Protocol</td>
<td>396</td>
</tr>
<tr>
<td>EFI_SERVICE_BINDING_PROTOCOL.CreateChild()</td>
<td>397</td>
</tr>
<tr>
<td>EFI_SERVICE_BINDING_PROTOCOL.DestroyChild()</td>
<td>401</td>
</tr>
<tr>
<td>11.7 EFI Platform to Driver Configuration Protocol</td>
<td>405</td>
</tr>
<tr>
<td>EFI_PLATFORM_TO_DRIVER_CONFIGURATION_PROTOCOL.Query()</td>
<td>405</td>
</tr>
<tr>
<td>EFI_PLATFORM_TO_DRIVER_CONFIGURATION_PROTOCOL.Response()</td>
<td>407</td>
</tr>
<tr>
<td>11.7.1 DMTF SM CLP ParameterTypeGuid</td>
<td>410</td>
</tr>
<tr>
<td>11.8 EFI Driver Supported EFI Version Protocol</td>
<td>412</td>
</tr>
<tr>
<td>11.9 EFI Driver Family Override Protocol</td>
<td>412</td>
</tr>
<tr>
<td>11.9.1 Overview</td>
<td>412</td>
</tr>
<tr>
<td>11.10 EFI Driver Health Protocol</td>
<td>414</td>
</tr>
<tr>
<td>11.10.1 UEFI Boot Manager Algorithms</td>
<td>422</td>
</tr>
<tr>
<td>11.10.2 UEFI Driver Algorithms</td>
<td>426</td>
</tr>
<tr>
<td>11.11 EFI Adapter Information Protocol</td>
<td>427</td>
</tr>
<tr>
<td>11.12 EFI Adapter Information Protocol Information Types</td>
<td>431</td>
</tr>
<tr>
<td>11.12.1 Network Media State</td>
<td>431</td>
</tr>
<tr>
<td>11.12.2 Network Boot</td>
<td>432</td>
</tr>
<tr>
<td>11.12.3 SAN MAC Address</td>
<td>433</td>
</tr>
<tr>
<td>11.12.4 IPV6 Support from UNDISRH</td>
<td>433</td>
</tr>
<tr>
<td>11.12.5 Network Media Type</td>
<td>434</td>
</tr>
</tbody>
</table>
### Table of Contents

11.12.6 Coherent Device Attribute Table (CDAT) Type ............................................................. 434

#### 12 Protocols — Console Support ..................................................................................... 436

12.1 Console I/O Protocol......................................................................................................... 436
  12.1.1 Overview ................................................................................................................. 436
  12.1.2 ConsoleIn Definition ............................................................................................. 436

12.2 Simple Text Input Ex Protocol ....................................................................................... 438
  EFI_SIMPLE_TEXT_INPUT_EX_PROTOCOL.............................................................................. 439
  EFI_SIMPLE_TEXT_INPUT_EX_PROTOCOL.Reset() .............................................................. 439
  EFI_SIMPLE_TEXT_INPUT_EX_PROTOCOL.ReadKeyStrokeEx() ........................................... 440
  EFI_SIMPLE_TEXT_INPUT_EX_PROTOCOL.SetState() ....................................................... 443
  EFI_SIMPLE_TEXT_INPUT_EX_PROTOCOL.RegisterKeyNotify() ........................................ 444
  EFI_SIMPLE_TEXT_INPUT_EX_PROTOCOL.UnregisterKeyNotify() .................................... 445

12.3 Simple Text Input Protocol ............................................................................................. 445
  EFI_SIMPLE_TEXT_INPUT_PROTOCOL ................................................................................. 446
  EFI_SIMPLE_TEXT_INPUT_PROTOCOL.Reset() .................................................................. 446
  EFI_SIMPLE_TEXT_INPUT_PROTOCOL.ReadKeyStroke() ..................................................... 447

12.4 Simple Text Output Protocol .......................................................................................... 449
  EFI_SIMPLE_TEXT_OUTPUT_PROTOCOL............................................................................. 451
  EFI_SIMPLE_TEXT_OUTPUT_PROTOCOL.Reset() ............................................................... 451
  EFI_SIMPLE_TEXT_OUTPUT_PROTOCOL.OutputString() .................................................... 451
  EFI_SIMPLE_TEXT_OUTPUT_PROTOCOL.TestString() ........................................................ 454
  EFI_SIMPLE_TEXT_OUTPUT_PROTOCOL.QueryMode() ...................................................... 455
  EFI_SIMPLE_TEXT_OUTPUT_PROTOCOL.SetMode() .......................................................... 456
  EFI_SIMPLE_TEXT_OUTPUT_PROTOCOL.SetAttribute() ................................................. 457
  EFI_SIMPLE_TEXT_OUTPUT_PROTOCOL.ClearScreen() .................................................... 458
  EFI_SIMPLE_TEXT_OUTPUT_PROTOCOL.SetCursorPosition() ....................................... 459
  EFI_SIMPLE_TEXT_OUTPUT_PROTOCOL.EnableCursor() ................................................. 460

12.5 Simple Pointer Protocol .................................................................................................. 460
  EFI_SIMPLE_POINTER_PROTOCOL..................................................................................... 461
  EFI_SIMPLE_POINTER_PROTOCOL.Reset() ...................................................................... 461
  EFI_SIMPLE_POINTER_PROTOCOL.GetState() ................................................................ 462

12.6 EFI Simple Pointer Device Paths .................................................................................... 464

12.7 Absolute Pointer Protocol .............................................................................................. 467
  EFI_ABSOLUTE_POINTER_PROTOCOL.............................................................................. 468
  EFI_ABSOLUTE_POINTER_PROTOCOL.Reset() ............................................................... 468
  EFI_ABSOLUTE_POINTER_PROTOCOL.GetState() .......................................................... 469

12.8 Serial I/O Protocol .......................................................................................................... 472
  EFI_SERIAL_IO_PROTOCOL ......................................................................................... 472
  12.8.1 Serial Device Identification .................................................................................... 476
  12.8.2 Serial Device Type GUIDs ..................................................................................... 476
# 13 Protocols — Media Access

## 13.1 Load File Protocol

- EFI_LOAD_FILE_PROTOCOL
- EFI_LOAD_FILE_PROTOCOL_LoadFile()

## 13.2 Load File 2 Protocol

- EFI_LOAD_FILE2_PROTOCOL
- EFI_LOAD_FILE2_PROTOCOL_LoadFile()

## 13.3 File System Format

- EFI_FILE_SYSTEM_PROTOCOL
  - EFI_FILE_SYSTEM_PROTOCOL_OpenVolume()

## 13.4 Simple File System Protocol

- EFI_SIMPLE_FILE_SYSTEM_PROTOCOL
- EFI_SIMPLE_FILE_SYSTEM_PROTOCOL_OpenVolume()

## 13.5 File Protocol

- EFI_FILE_PROTOCOL
  - EFI_FILE_PROTOCOL_Open()
  - EFI_FILE_PROTOCOL_Close()
  - EFI_FILE_PROTOCOL_Delete()
  - EFI_FILE_PROTOCOL_Read()
  - EFI_FILE_PROTOCOL_Write()
  - EFI_FILE_PROTOCOLGetPosition()
  - EFI_FILE_PROTOCOL_GetPosition()
  - EFI_FILE_PROTOCOLGetPosition()
  - EFI_FILE_PROTOCOLGetPosition()
  - EFI_FILE_PROTOCOLGetPosition()
  - EFI_FILE_PROTOCOLGetPosition()
  - EFI_FILE_PROTOCOLGetPosition()
  - EFI_FILE_PROTOCOLGetPosition()
  - EFI_FILE_PROTOCOLGetPosition()
  - EFI_FILE_PROTOCOLGetPosition()
  - EFI_FILE_PROTOCOLGetPosition()
  - EFI_FILE_PROTOCOLGetPosition()
  - EFI_FILE_PROTOCOLGetPosition()
  - EFI_FILE_PROTOCOLGetPosition()
  - EFI_FILE_PROTOCOLGetPosition()
  - EFI_FILE_PROTOCOLGetPosition()
  - EFI_FILE_PROTOCOLGetPosition()
  - EFI_FILE_PROTOCOLGetPosition()
  - EFI_FILE_PROTOCOLGetPosition()
  - EFI_FILE_PROTOCOLGetPosition()
  - EFI_FILE_PROTOCOLGetPosition()
  - EFI_FILE_PROTOCOLGetPosition()
  - EFI_FILE_PROTOCOLGetPosition()
  - EFI_FILE_PROTOCOLGetPosition()
  - EFI_FILE_PROTOCOLGetPosition()
  - EFI_FILE_PROTOCOLGetPosition()
  - EFI_FILE_PROTOCOLGetPosition()
  - EFI_FILE_PROTOCOLGetPosition()
  - EFI_FILE_PROTOCOLGetPosition()
  - EFI_FILE_PROTOCOLGetPosition()
  - EFI_FILE_PROTOCOLGetPosition()
  - EFI_FILE_PROTOCOLGetPosition()
  - EFI_FILE_PROTOCOLGetPosition()
  - EFI_FILE_PROTOCOLGetPosition()
  - EFI_FILE_PROTOCOLGetPosition()
  - EFI_FILE_PROTOCOLGetPosition()
  - EFI_FILE_PROTOCOLGetPosition()
  - EFI_FILE_PROTOCOLGetPosition()
  - EFI_FILE_PROTOCOLGetPosition()
  - EFI_FILE_PROTOCOLGetPosition()
  - EFI_FILE_PROTOCOLGetPosition()
  - EFI_FILE_PROTOCOLGetPosition()
  - EFI_FILE_PROTOCOLGetPosition()
  -EFI_FILE_PROTOCOLGetPosition()
  - EFI_FILE_PROTOCOLGetPosition()
  - EFI_FILE_PROTOCOLGetPosition()
  - EFI_FILE_PROTOCOLGetPosition()
  - EFI_FILE_PROTOCOLGetPosition()
  - EFI_FILE_PROTOCOLGetPosition()
  - EFI_FILE_PROTOCOLGetPosition()
  - EFI_FILE_PROTOCOLGetPosition()
  - EFI_FILE_PROTOCOLGetPosition()
  - EFI_FILE_PROTOCOLGetPosition()
  - EFI_FILE_PROTOCOLGetPosition()
  - EFI_FILE_PROTOCOLGetPosition()
  - EFI_FILE_PROTOCOLGetPosition()
  - EFI_FILE_PROTOCOLGetPosition()
  - EFI_FILE_PROTOCOLGetPosition()
  - EFI_FILE_PROTOCOLGetPosition()
  - EFI_FILE_PROTOCOLGetPosition()
  - EFI_FILE_PROTOCOLGetPosition()
  - EFI_FILE_PROTOCOLGetPosition()
  - EFI_FILE_PROTOCOLGetPosition()
  - EFI_FILE_PROTOCOLGetPosition()
  - EFI_FILE_PROTOCOLGetPosition()
  - EFI_FILE_PROTOCOLGetPosition()
  - EFI_FILE_PROTOCOLGetPosition()
  - EFI_FILE_PROTOCOLGetPosition()
  - EFI_FILE_PROTOCOLGetPosition()
  - EFI_FILE_PROTOCOLGetPosition()
  - EFI_FILE_PROTOCOLGetPosition()
  - EFI_FILE_PROTOCOLGetPosition()
  - EFI_FILE_PROTOCOLGetPosition()
  - EFI_FILE_PROTOCOLGetPosition()
  - EFI_FILE_PROTOCOLGetPosition()
  - EFI_FILE_PROTOCOLGetPosition()
  - EFI_FILE_PROTOCOLGetPosition()
  - EFI_FILE_PROTOCOLGetPosition()
  - EFI_FILE_PROTOCOLGetPosition()
  - EFI_FILE_PROTOCOLGetPosition()
  - EFI_FILE_PROTOCOLGetPosition()
  - EFI_FILE_PROTOCOLGetPosition()
  - EFI_FILE_PROTOCOLGetPosition()
  - EFI_FILE_PROTOCOLGetPosition()
  - EFI_FILE_PROTOCOLGetPosition()
  - EFI_FILE_PROTOCOLGetPosition()
  - EFI_FILE_PROTOCOLGetPosition()
  - EFI_FILE_PROTOCOLGetPosition()
  - EFI_FILE_PROTOCOLGetPosition()
  - EFI_FILE_PROTOCOLGetPosition()
  - EFI_FILE_PROTOCOLGetPosition()
  - EFI_FILE_PROTOCOLGetPosition()
  - EFI_FILE_PROTOCOLGetPosition()
  - EFI_FILE_PROTOCOLGetPosition()
  - EFI_FILE_PROTOCOLGetPosition()
  - EFI_FILE_PROTOCOLGetPosition()
  - EFI_FILE_PROTOCOLGetPosition()
  - EFI_FILE_PROTOCOLGetPosition()
  - EFI_FILE_PROTOCOLGetPosition()
  - EFI_FILE_PROTOCOLGetPosition()
  - EFI_FILE_PROTOCOLGetPosition()
  - EFI_FILE_PROTOCOLGetPosition()
  - EFI_FILE_PROTOCOLGetPosition()
  - EFI_FILE_PROTOCOLGetPosition()
  - EFI_FILE_PROTOCOLGetPosition()
  - EFI_FILE_PROTOCOLGetPosition()
  - EFI_FILE_PROTOCOLGetPosition()
  - EFI_FILE_PROTOCOLGetPosition()
  - EFI_FILE_PROTOCOLGetPosition()
  - EFI_FILE_PROTOCOLGetPosition()
  - EFI_FILE_PROTOCOLGetPosition()
  - EFI_FILE_PROTOCOLGetPosition()
  - EFI_FILE_PROTOCOLGetPosition()
  - EFI_FILE_PROTOCOLGetPosition()
  - EFI_FILE_PROTOCOLGetPosition()
  - EFI_FILE_PROTOCOLGetPosition()
  - EFI_FILE_PROTOCOLGetPosition()
  - EFI_FILE_PROTOCOLGetPosition()
  - EFI_FILE_PROTOCOLGetPosition()
  - EFI_FILE_PROTOCOLGetPosition()
  - EFI_FILE_PROTOCOLGetPosition()
  - EFI_FILE_PROTOCOLGetPosition()
  - EFI_FILE_PROTOCOLGetPosition()
  - EFI_FILE_PROTOCOLGetPosition()
  - EFI_FILE_PROTOCOLGetPosition()
  - EFI_FILE_PROTOCOLGetPosition()
  - EFI_FILE_PROTOCOLGetPosition()
  - EFI_FILE_PROTOCOLGetPosition()
  - EFI_FILE_PROTOCOLGetPosition()
  - EFI_FILE_PROTOCOLGetPosition()
  - EFI_FILE_PROTOCOLGetPosition()
  - EFI_FILE_PROTOCOLGetPosition()
  - EFI_FILE_PROTOCOLGetPosition()
  - EFI_FILE_PROTOCOLGetPosition()
  - EFI_FILE_PROTOCOLGetPosition()
  - EFI_FILE_PROTOCOLGetPosition()
  - EFI_FILE_PROTOCOLGetPosition()
  - EFI_FILE_PROTOCOLGetPosition()
  - EFI_FILE_PROTOCOLGetPosition()
  - EFI_FILE_PROTOCOLGetPosition()
  - EFI_FILE_PROTOCOLGetPosition()
  - EFI_FILE_PROTOCOLGetPosition()
  - EFI_FILE_PROTOCOLGetPosition()
  - EFI_FILE_PROTOCOLGetPosition()
  - EFI_FILE_PROTOCOLGetPosition()
  - EFI_FILE_PROTOCOLGetPosition()
  - EFI_FILE_PROTOCOLGetPosition()
  - EFI_FILE_PROTOCOLGetPosition()
  - EFI_FILE_PROTOCOLGetPosition()
  - EFI_FILE_PROTOCOLGetPosition()
  - EFI_FILE_PROTOCOLGetPosition()
  - EFI_FILE_PROTOCOLGetPosition()
  - EFI_FILE_PROTOCOLGetPosition()
  - EFI_FILE_PROTOCOLGetPosition()
  - EFI_FILE_PROTOCOLGetPosition()
  - EFI_FILE_PROTOCOLGetPosition()
  - EFI_FILE_PROTOCOLGetPosition()
  - EFI_FILE_PROTOCOLGetPosition()
  - EFI_FILE_PROTOCOLGetPosition()
  - EFI_FILE_PROTOCOLGetPosition()
  - EFI_FILE_PROTOCOLGetPosition()
  - EFI_FILE_PROTOCOLGetPosition()
  - EFI_FILE_PROTOCOLGetPosition()
  - EFI_FILE_PROTOCOLGetPosition()
  - EFI_FILE_PROTOCOLGetPosition()
  - EFI_FILE_PROTOCOLGetPosition()
  - EFI_FILE_PROTOCOLGetPosition()
  - EFI_FILE_PROTOCOLGetPosition()
  - EFI_FILE_PROTOCOLGetPosition()
  - EFI_FILE_PROTOCOLGetPosition()
  - EFI_FILE_PROTOCOLGetPosition()
  - EFI_FILE_PROTOCOLGetPosition()
  - EFI_FILE_PROTOCOLGetPosition()
  - EFI_FILE_PROTOCOLGetPosition()
  - EFI_FILE_PROTOCOLGetPosition()
  - EFI_FILE_PROTOCOLGetPosition()
  - EFI_FILE_PROTOCOLGetPosition()
  - EFI_FILE_PROTOCOLGetPosition()
  - EFI_FILE_PROTOCOLGetPosition()
  - EFI_FILE_PROTOCOLGetPosition()
  - EFI_FILE_PROTOCOLGetPosition()
  - EFI_FILE_PROTOCOLGetPosition()
  - EFI_FILE_PROTOCOLGetPosition()
  - EFI_FILE_PROTOCOLGetPosition()
  - EFI_FILE_PROTOCOLGetPosition()
  - EFI_FILE_PROTOCOLGetPosition()
  - EFI_FILE_PROTOCOLGetPosition()
  - EFI_FILE_PROTOCOLGetPosition()
  - EFI_FILE_PROTOCOLGetPosition()
  - EFI_FILE_PROTOCOLGetPosition()
  - EFI_FILE_PROTOCOLGetPosition()
  - EFI_FILE_PROTOCOLGetPosition()
  - EFI_FILE_PROTOCOLGetPosition()
  - EFI_FILE_PROTOCOLGetPosition()
  - EFI_FILE_PROTOCOLGetPosition()
  - EFI_FILE_PROTOCOLGetPosition()
  - EFI_FILE_PROTOCOLGetPosition()
  - EFI_FILE_PROTOCOLGetPosition()
  - EFI_FILE_PROTOCOLGetPosition()
  - EFI_FILE_PROTOCOLGetPosition()
  - EFI_FILE_PROTOCOLGetPosition()
13.6.1 Tape I/O Support .................................................................................................. 533
13.6.2 Tape I/O Protocol .............................................................................................. 533
   EFI_TAPE_IO_PROTOCOL ................................................................. 533
   EFI_TAPE_IO_PROTOCOL.TapeRead() .............................................. 535
   EFI_TAPE_IO_PROTOCOL.TapeWrite() ............................................. 536
   EFI_TAPE_IO_PROTOCOL.TapeRewind() ........................................... 537
   EFI_TAPE_IO_PROTOCOL.TapeSpace() ............................................ 538
   EFI_TAPE_IO_PROTOCOL.TapeWriteFM() .......................................... 539
   EFI_TAPE_IO_PROTOCOL.TapeReset() .............................................. 540
13.6.3 Tape Header Format ......................................................................................... 541
13.7 Disk I/O Protocol ............................................................................................... 542
   EFI_DISK_IO_PROTOCOL ................................................................. 543
   EFI_DISK_IO_PROTOCOL.ReadDisk() .................................................. 544
   EFI_DISK_IO_PROTOCOL.WriteDisk() ............................................... 545
13.8 Disk I/O 2 Protocol ............................................................................................ 546
   EFI_DISK_IO2_PROTOCOL ................................................................. 546
   EFI_DISK_IO2_PROTOCOL.Cancel() .................................................. 547
   EFI_DISK_IO2_PROTOCOL.ReadDiskEx() ............................................. 548
   EFI_DISK_IO2_PROTOCOL.WriteDiskEx() .......................................... 549
   EFI_DISK_IO2_PROTOCOL.FlushDiskEx() ........................................... 551
13.9 Block I/O Protocol ............................................................................................. 552
   EFI_BLOCK_IO_PROTOCOL ................................................................. 552
   EFI_BLOCK_IO_PROTOCOL.Reset() .................................................... 556
   EFI_BLOCK_IO_PROTOCOL.ReadBlocks() .......................................... 556
   EFI_BLOCK_IO_PROTOCOL.WriteBlocks() .......................................... 558
   EFI_BLOCK_IO_PROTOCOL.FlushBlocks() .......................................... 559
13.10 Block I/O 2 Protocol ......................................................................................... 559
   EFI_BLOCK_IO2_PROTOCOL ............................................................... 560
   EFI_BLOCK_IO2_PROTOCOL.Reset() ................................................... 560
   EFI_BLOCK_IO2_PROTOCOL.ReadBlocksEx() ..................................... 561
   EFI_BLOCK_IO2_PROTOCOL.WriteBlocksEx() ..................................... 563
   EFI_BLOCK_IO2_PROTOCOL.FlushBlocksEx() ..................................... 565
13.11 Inline Cryptographic Interface Protocol .......................................................... 566
   EFI_BLOCK_IO_CRYPTO_PROTOCOL ................................................. 566
   EFI_BLOCK_IO_CRYPTO_PROTOCOL.Reset() ...................................... 570
   EFI_BLOCK_IO_CRYPTO_PROTOCOL.GetCapabilities() ....................... 571
   EFI_BLOCK_IO_CRYPTO_PROTOCOL.SetConfiguration() ................... 572
   EFI_BLOCK_IO_CRYPTO_PROTOCOL.GetConfiguration() ................... 574
   EFI_BLOCK_IO_CRYPTO_PROTOCOL.ReadExtended() ......................... 575
   EFI_BLOCK_IO_CRYPTO_PROTOCOL.WriteExtended() ......................... 577
   EFI_BLOCK_IO_CRYPTO_PROTOCOL.FlushBlocks() ............................. 579
13.12 Erase Block Protocol ....................................................................................... 580
   EFI_ERASE_BLOCK_PROTOCOL ....................................................... 580
13.13 ATA Pass Thru Protocol ................................................................................... 582
   EFIATA_PASS_THRU_PROTOCOL ..................................................... 582
   EFI ATA_PASS_THRU_PROTOCOL.PassThru() .................................. 586
13.14 Storage Security Command Protocol ................................................................. 599
EFI_STORAGE_SECURITY_COMMAND_PROTOCOL .................................................. 600
EFI_STORAGE_SECURITY_COMMAND_PROTOCOL.ReceiveData() ........................... 601
EFI_STORAGE_SECURITY_COMMAND_PROTOCOL.SendData() ............................... 603
13.15 NVM Express Pass Through Protocol ............................................................. 605
EFI_NVM_EXPRESS_PASS_THRU_PROTOCOL ................................................ 605
EFI_NVM_EXPRESS_PASS_THRU_PROTOCOL.PassThru() .................................... 608
EFI_NVM_EXPRESS_PASS_THRU_PROTOCOL.GetNextNamespace() ..................... 612
EFI_NVM_EXPRESS_PASS_THRU_PROTOCOL.BuildDevicePath() ......................... 613
EFI_NVM_EXPRESS_PASS_THRU_PROTOCOL.GetNamespace() ............................. 614
13.16 SD MMC Pass Thru Protocol ............................................................................. 615
EFI_SD_MMC_PASS_THRU_PROTOCOL ............................................................... 615
EFI_SD_MMC_PASS_THRU_PROTOCOL.PassThru() ............................................ 616
EFI_SD_MMC_PASS_THRU_PROTOCOL.GetNextSlot() .......................................... 619
EFI_SD_MMC_PASS_THRU_PROTOCOL.BuildDevicePath() ................................. 620
EFI_SD_MMC_PASS_THRU_PROTOCOL.GetSlotNumber() .................................... 621
EFI_SD_MMC_PASS_THRU_PROTOCOL.ResetDevice() ........................................ 621
13.17 RAM Disk Protocol ......................................................................................... 623
EFI_RAM_DISK_PROTOCOL ............................................................................. 623
EFI_RAM_DISK_PROTOCOL.Register() ............................................................. 623
EFI_RAM_DISK_PROTOCOL.Unregister() ......................................................... 625
13.18 Partition Information Protocol ......................................................................... 626
13.19 NVDIMM Label Protocol ................................................................................ 628
EFI_NVDIMM_LABEL_PROTOCOL.LabelStorageInformation() ........................ 628
EFI_NVDIMM_LABEL_PROTOCOL.LabelStorageRead() ....................................... 629
EFI_NVDIMM_LABEL_PROTOCOL.LabelStorageWrite() ....................................... 630
13.20 EFI UFS Device Config Protocol .................................................................... 645
EFI_UFS_DEVICE_CONFIG_PROTOCOL ............................................................... 645
EFI_UFS_DEVICE_CONFIG_PROTOCOL.RwUfsDescriptor() .............................. 646
EFI_UFS_DEVICE_CONFIG_PROTOCOL.RwUvsFlag() ......................................... 647
EFI_UFS_DEVICE_CONFIG_PROTOCOL.RwUvsAttribute() .................................. 647
14 Protocols — PCI Bus Support ............................................................................ 649
14.1 PCI Root Bridge I/O Support ............................................................................ 649
14.1.1 PCI Root Bridge I/O Overview .................................................................... 649
14.2 PCI Root Bridge I/O Protocol ......................................................................... 654
EFI_PCI_ROOT_BRIDGE_IO_PROTOCOL ....................................................... 654
EFI_PCI_ROOT_BRIDGE_IO_PROTOCOL.PollMem() ......................................... 661
EFI_PCI_ROOT_BRIDGE_IO_PROTOCOL.PollIo() ................................................. 662
EFI_PCI_ROOT_BRIDGE_IO_PROTOCOL.Mem.Read() ......................................... 664
EFI_PCI_ROOT_BRIDGE_IO_PROTOCOL.Mem.Write() ......................................... 664
14.2.1 PCI Root Bridge Device Paths .......................................................... 680
14.3 PCI Driver Model .................................................................................... 683
  14.3.1 PCI Driver Initialization .................................................................... 683
  14.3.2 PCI Bus Drivers ................................................................................ 685
  14.3.3 PCI Device Drivers .......................................................................... 690
14.4 EFI PCI I/O Protocol .................................................................................. 691
  EFI_PCI_ROOT_BRIDGE_IO_PROTOCOL.Io.Read() ............................................ 665
  EFI_PCI_ROOT_BRIDGE_IO_PROTOCOL.Io.Write() ............................................ 666
  EFI_PCI_ROOT_BRIDGE_IO_PROTOCOL.Pci.Read() ............................................. 666
  EFI_PCI_ROOT_BRIDGE_IO_PROTOCOL.Pci.Write() .......................................... 667
  EFI_PCI_ROOT_BRIDGE_IO_PROTOCOL.CopyMem() .......................................... 668
  EFI_PCI_ROOT_BRIDGE_IO_PROTOCOL.Map() ................................................. 670
  EFI_PCI_ROOT_BRIDGE_IO_PROTOCOL.Unmap() ............................................. 671
  EFI_PCI_ROOT_BRIDGE_IO_PROTOCOL.AllocateBuffer() .................................. 672
  EFI_PCI_ROOT_BRIDGE_IO_PROTOCOL.FreeBuffer() ....................................... 674
  EFI_PCI_ROOT_BRIDGE_IO_PROTOCOL.Flush() .............................................. 674
  EFI_PCI_ROOT_BRIDGE_IO_PROTOCOL.GetAttributes() ................................... 675
  EFI_PCI_ROOT_BRIDGE_IO_PROTOCOL.SetAttributes() ................................... 676
  EFI_PCI_ROOT_BRIDGE_IO_PROTOCOL.Configuration() .................................... 678
14.4.1 PCI Device Paths .................................................................................. 724
14.4.2 PCI Option ROMs .................................................................................. 726
14.4.3 Nonvolatile Storage .............................................................................. 737
14.4.4 PCI Hot-Plug Events ............................................................................ 737
15 Protocols — SCSI Driver Models and Bus Support .................................... 738
  15.1 SCSI Driver Model Overview ............................................................... 738
  15.2 SCSI Bus Drivers .................................................................................. 738
    15.2.1 Driver Binding Protocol for SCSI Bus Drivers ................................. 739
15.2.2 SCSI Enumeration ................................................................. 740
15.3 SCSI Device Drivers ............................................................... 740
  15.3.1 Driver Binding Protocol for SCSI Device Drivers ................. 740
15.4 EFI SCSI I/O Protocol ............................................................... 740
  EFI_SCSI_IO_PROTOCOL .......................................................... 741
    EFI_SCSI_IO_PROTOCOL.GetDeviceType() .................................. 742
    EFI_SCSI_IO_PROTOCOL.GetDeviceLocation() ............................... 743
    EFI_SCSI_IO_PROTOCOL.ResetBus() ............................................. 744
    EFI_SCSI_IO_PROTOCOL.ResetDevice() ......................................... 745
    EFI_SCSI_IO_PROTOCOL.ExecuteScsiCommand() ............................ 746
15.5 SCSI Device Paths ................................................................. 750
  15.5.1 SCSI Device Path Example ................................................. 750
  15.5.2 ATAPI Device Path Example ............................................... 751
  15.5.3 Fibre Channel Device Path Example ..................................... 752
  15.5.4 InfiniBand Device Path Example ......................................... 753
15.6 SCSI Pass Thru Device Paths ................................................... 754
15.7 Extended SCSI Pass Thru Protocol ............................................. 757
  EFI_EXT_SCSI_PASS_THRU_PROTOCOL ........................................ 757
    EFI_EXT_SCSI_PASS_THRU_PROTOCOL.PassThru() .......................... 760
    EFI_EXT_SCSI_PASS_THRU_PROTOCOL.GetNextTargetLun() .............. 766
    EFI_EXT_SCSI_PASS_THRU_PROTOCOL.BuildDevicePath() .................. 768
    EFI_EXT_SCSI_PASS_THRU_PROTOCOL.GetTargetLun() ...................... 769
    EFI_EXT_SCSI_PASS_THRU_PROTOCOL.ResetChannel() ...................... 770
    EFI_EXT_SCSI_PASS_THRU_PROTOCOL.ResetTargetLun() .................... 771
    EFI_EXT_SCSI_PASS_THRU_PROTOCOL.GetNextTarget() ..................... 772
16 Protocols — iSCSI Boot ............................................................ 774
16.1 Overview ................................................................................ 774
  16.1.1 iSCSI UEFI Driver Layering ............................................... 774
16.2 EFI iSCSI Initiator Name Protocol .......................................... 774
  EFI_ISCSI_INITIATOR_NAME_PROTOCOL .................................... 774
    EFI_ISCSI_INITIATOR_NAME_PROTOCOL.Get() ............................... 775
    EFI_ISCSI_INITIATOR_NAME_PROTOCOL.Set() ............................... 776
17 Protocols — USB Support ......................................................... 778
17.1 USB2 Host Controller Protocol ............................................... 778
  17.1.1 USB Host Controller Protocol Overview ............................... 778
    EFI_USB2_HC_PROTOCOL ..................................................... 778
      EFI_USB2_HC_PROTOCOL.GetCapability() .................................. 780
      EFI_USB2_HC_PROTOCOL.Reset() ............................................. 781
      EFI_USB2_HC_PROTOCOL.GetState() .......................................... 783
      EFI_USB2_HC_PROTOCOL.SetState() .......................................... 784
      EFI_USB2_HC_PROTOCOL.ControlTransfer() ................................. 785
      EFI_USB2_HC_PROTOCOL.BulkTransfer() ..................................... 788
      EFI_USB2_HC_PROTOCOL.AsyncInterruptTransfer() ....................... 790
      EFI_USB2_HC_PROTOCOL.SyncInterruptTransfer() ......................... 793
      EFI_USB2_HC_PROTOCOL.IsochronousTransfer() ............................ 795
17.2 USB Driver Model ........................................................................................................ 806
17.2.1 Scope .................................................................................................................... 806
17.2.2 USB Bus Driver ........................................................................................................ 807
17.2.3 USB Device Driver .................................................................................................... 808
17.2.4 USB I/O Protocol ....................................................................................................... 809
17.3 USB Function Protocol .................................................................................................. 830
18 Protocols — Debugger Support......................................................................................856
18.1 Overview .................................................................................................................... 856
18.2 EFI Debug Support Protocol ..................................................................................... 856
18.2.1 EFI Debug Support Protocol Overview ................................................................. 856
18.3 EFI Debugport Protocol .................................................. 875
  18.3.1 EFI Debugport Overview ........................................... 875
  18.3.2 Debugport Device Path ............................................. 879
  18.3.3 EFI Debugport Variable ........................................... 880
18.4 EFI Debug Support Table .................................................. 881
  18.4.1 Overview ............................................................... 881
  18.4.2 EFI System Table Location ........................................ 882
  18.4.3 EFI Image Info ....................................................... 882
19 Protocols — Compression Algorithm Specification ............... 885
  19.1 Algorithm Overview ................................................... 885
  19.2 Data Format ............................................................ 886
    19.2.1 Bit Order ........................................................... 886
    19.2.2 Overall Structure ................................................ 887
    19.2.3 Block Structure .................................................... 887
  19.3 Compressor Design ................................................... 890
    19.3.1 Overall Process ................................................... 890
    19.3.2 String Info Log .................................................... 891
    19.3.3 Huffman Code Generation .................................... 894
  19.4 Decompressor Design ................................................ 896
  19.5 Decompress Protocol ................................................ 897
    EFI_DEC.compress_PROTOCOL ........................................... 897
    EFI_DEC.compress_PROTOCOL.GetInfo() .............................. 897
    EFI_DEC.compress_PROTOCOL.Decompress() .......................... 899
20 Protocols — ACPI Protocols .................................................. 901
    EFI_ACPI_TABLE_PROTOCOL ........................................... 901
    EFI_ACPI_TABLE_PROTOCOL.InstallAcpiTable() ...................... 901
    EFI_ACPI_TABLE_PROTOCOL.UninstallAcpiTable() .................. 902
21 Protocols — String Services .................................................. 904
  21.1 Unicode Collation Protocol .......................................... 904
    EFI_UNICODE_COLLATION_PROTOCOL .................................. 904
    EFI_UNICODE_COLLATION_PROTOCOL.StriColl() ..................... 905
    EFI_UNICODE_COLLATION_PROTOCOL.MetaiMatch() .................. 906
    EFI_UNICODE_COLLATION_PROTOCOL.StrLwr() ....................... 907
    EFI_UNICODE_COLLATION_PROTOCOL.StrUpr() ....................... 907
## 22 EFI Byte Code Virtual Machine

### 22.1 Overview
- Processor Architecture Independence
- OS Independent
- EFI Compliant
- Coexistence of Legacy Option ROMs
- Relocatable Image
- Size Restrictions Based on Memory Available

### 22.2 Memory Ordering

### 22.3 Virtual Machine Registers

### 22.4 Natural Indexing
- Sign Bit
- Bits Assigned to Natural Units
- Constant
- Natural Units

### 22.5 EBC Instruction Operands
- Direct Operands
- Indirect Operands
- Indirect with Index Operands
- Immediate Operands

### 22.6 EBC Instruction Syntax

### 22.7 Instruction Encoding
- Instruction Opcode Byte Encoding
- Instruction Operands Byte Encoding
- Index/Immediate Data Encoding

### 22.8 EBC Instruction Set
- ADD
- AND
- ASHR
- BREAK
- CALL
- CMP
- CMPI
- DIV
- DIVU
- EXTNDB
- EXTNDW
- JMP
<table>
<thead>
<tr>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>22.9 Runtime and Software Conventions</td>
<td>962</td>
</tr>
<tr>
<td>22.9.1 Calling Outside VM</td>
<td>962</td>
</tr>
<tr>
<td>22.9.2 Calling Inside VM</td>
<td>962</td>
</tr>
<tr>
<td>22.9.3 Parameter Passing</td>
<td>963</td>
</tr>
<tr>
<td>22.9.4 Return Values</td>
<td>963</td>
</tr>
<tr>
<td>22.9.5 Binary Format</td>
<td>963</td>
</tr>
<tr>
<td>22.10 Architectural Requirements</td>
<td>963</td>
</tr>
<tr>
<td>22.10.1 EBC Image Requirements</td>
<td>963</td>
</tr>
<tr>
<td>22.10.2 EBC Execution Interfacing Requirements</td>
<td>963</td>
</tr>
<tr>
<td>22.10.3 Interfacing Function Parameters Requirements</td>
<td>964</td>
</tr>
<tr>
<td>22.10.4 Function Return Requirements</td>
<td>964</td>
</tr>
<tr>
<td>22.10.5 Function Return Values Requirements</td>
<td>964</td>
</tr>
<tr>
<td>22.11 EBC Interpreter Protocol</td>
<td>964</td>
</tr>
<tr>
<td>EFI_EBC_PROTOCOL</td>
<td>964</td>
</tr>
<tr>
<td>EFI_EBC_PROTOCOL.CreateThunk()</td>
<td>965</td>
</tr>
<tr>
<td>EFI_EBC_PROTOCOL.UnloadImage()</td>
<td>966</td>
</tr>
<tr>
<td>EFI_EBC_PROTOCOL.RegisterICacheFlush()</td>
<td>967</td>
</tr>
<tr>
<td>EFI_EBC_PROTOCOL.GetVersion()</td>
<td>968</td>
</tr>
<tr>
<td>22.12 EBC Tools</td>
<td>969</td>
</tr>
<tr>
<td>22.12.1 EBC C Compiler</td>
<td>969</td>
</tr>
<tr>
<td>22.12.2 C Coding Convention</td>
<td>969</td>
</tr>
<tr>
<td>22.12.3 EBC Interface Assembly Instructions</td>
<td>969</td>
</tr>
<tr>
<td>Section</td>
<td>Page</td>
</tr>
<tr>
<td>---------</td>
<td>------</td>
</tr>
<tr>
<td>DECLARE_LENGTH</td>
<td>1001</td>
</tr>
<tr>
<td>23.3 Delivering Capsules Containing Updates to Firmware Management Protocol</td>
<td>1002</td>
</tr>
<tr>
<td>23.3.1 EFI_FIRMWARE_MANAGEMENT_CAPSULE_ID_GUID</td>
<td>1002</td>
</tr>
<tr>
<td>23.3.2 DEFINED Firmware Management Protocol DATA Capsule STRUCTURE</td>
<td>1002</td>
</tr>
<tr>
<td>23.3.3 Firmware Processing of the Capsule Identified by EFI_FIRMWARE_MANAGEMENT_CAPSULE_ID_GUID</td>
<td>1007</td>
</tr>
<tr>
<td>23.4 EFI System Resource Table</td>
<td>1009</td>
</tr>
<tr>
<td>EFSYSTEMRESOURCE_TABLE</td>
<td>1009</td>
</tr>
<tr>
<td>23.4.1 Adding and Removing Devices from the ESRT</td>
<td>1011</td>
</tr>
<tr>
<td>23.4.2 ESRT and Firmware Management Protocol</td>
<td>1012</td>
</tr>
<tr>
<td>23.4.3 Mapping Firmware Management Protocol Descriptors to ESRT Entries</td>
<td>1012</td>
</tr>
<tr>
<td>23.5 Delivering Capsule Containing JSON payload</td>
<td>1013</td>
</tr>
<tr>
<td>23.5.1 EFI_JSON_CAPSULE_ID_GUID</td>
<td>1013</td>
</tr>
<tr>
<td>23.5.2 Defined JSON Capsule Data Structure</td>
<td>1014</td>
</tr>
<tr>
<td>23.5.3 Firmware Processing of the Capsule Identified by EFI_JSON_CAPSULE_ID_GUID</td>
<td>1015</td>
</tr>
</tbody>
</table>

24 Network Protocols — SNP, PXE, BIS and HTTP Boot

24.1 Simple Network Protocol | 1017
| EFI_SIMPLE_NETWORK_PROTOCOL | 1017 |
| EFI_SIMPLE_NETWORK.Start() | 1021 |
| EFI_SIMPLE_NETWORK.Stop() | 1022 |
| EFI_SIMPLE_NETWORK.Initialize() | 1023 |
| EFI_SIMPLE_NETWORK.Reset() | 1024 |
| EFI_SIMPLE_NETWORK.Shutdown() | 1025 |
| EFI_SIMPLE_NETWORK.ReceiveFilters() | 1025 |
| EFI_SIMPLE_NETWORK.StationAddress() | 1028 |
| EFI_SIMPLE_NETWORK.Statistics() | 1029 |
| EFI_SIMPLE_NETWORK.MCastIPtoMAC() | 1032 |
| EFI_SIMPLE_NETWORK.NvData() | 1033 |
| EFI_SIMPLE_NETWORK.GetStatus() | 1034 |
| EFI_SIMPLE_NETWORK.Transmit() | 1035 |
| EFI_SIMPLE_NETWORK.Receive() | 1037 |

24.2 Network Interface Identifier Protocol | 1038
| ENETWORKINTERFACEIDENTIFIER_PROTOCOL | 1039 |

24.3 PXE Base Code Protocol | 1041
| EPIXEBASE_CODE_PROTOCOL | 1041 |
| EPIXEBASE_CODE_PROTOCOL.Start() | 1051 |
| EPIXEBASE_CODE_PROTOCOL.Stop() | 1053 |
| EPIXEBASE_CODE_PROTOCOL.Dhcp() | 1054 |
| EPIXEBASE_CODE_PROTOCOL.Discover() | 1055 |
| EPIXEBASE_CODE_PROTOCOL.Mtftp() | 1055 |
| EPIXEBASE_CODE_PROTOCOL.UdpWrite() | 1062 |
| EPIXEBASE_CODE_PROTOCOL.UdpRead() | 1064 |
| EPIXEBASE_CODE_PROTOCOL.SetIpFilter() | 1067 |
| EPIXEBASE_CODE_PROTOCOL.Arp() | 1068 |
| EPIXEBASE_CODE_PROTOCOL.SetParameters() | 1070 |
| EPIXEBASE_CODE_PROTOCOL.SetStationIp() | 1071 |
24.3.1 Netboot6.................................................................1073
24.4 PXE Base Code Callback Protocol.................................................................1079
  EFI_PXE_BASE_CODE_CALLBACK_PROTOCOL.SetPackets()...............................1080
  EFI_PXE_BASE_CODE_CALLBACK_PROTOCOL.Callback().................................1080
24.5 Boot Integrity Services Protocol ....................................................................1082
  EFI_BIS_PROTOCOL............................................................................................1082
  EFI_BIS_PROTOCOL.Initialize() .......................................................................1084
  EFI_BIS_PROTOCOL.Shutdown() ......................................................................1087
  EFI_BIS_PROTOCOL.Free().................................................................................1088
  EFI_BIS_PROTOCOL.GetBootObjectAuthorizationCertificate()........................1089
  EFI_BIS_PROTOCOL.GetBootObjectAuthorizationCheckFlag().........................1090
  EFI_BIS_PROTOCOL.SetSignatureInfo()..........................................................1091
  EFI_BIS_PROTOCOL.GetBootObjectAuthorizationUpdateToken().....................1092
  EFI_BIS_PROTOCOL.VerifyBootObject()..........................................................1104
  EFI_BIS_PROTOCOL.VerifyObjectWithCredential().........................................1110
24.6 DHCP options for ISCSI on IPV6 .................................................................1116
24.7 HTTP Boot .................................................................................................1117
  24.7.1 Boot from URL......................................................................................1117
  24.7.2 Concept configuration for a typical HTTP Boot scenario .......................1118
  24.7.3 Protocol Layout for UEFI HTTP Boot Client concept configuration for a typical HTTP Boot scenario..................................................................................................................1120
  24.7.4 Concept of Message Exchange in a typical HTTP Boot scenario (IPv4 in Corporate Environment)...............................................................1122
  24.7.5 Concept of Message Exchange in HTTP Boot scenario (IPv6)...............1126
  24.7.6 EFI HTTP Boot Callback Protocol......................................................1128
    EFI_HTTP_BOOT_CALLBACK_PROTOCOL.setPackets().....................................1128
    EFI_HTTP_BOOT_CALLBACK_PROTOCOL.Callback()........................................1128

25 Network Protocols — Managed Network ...........................................................1131
  25.1 EFI Managed Network Protocol..............................................................1131
    EFI_MANAGED_NETWORK_SERVICE_BINDING_PROTOCOL..............................1131
    EFI_MANAGED_NETWORK_PROTOCOL................................................................1131
    EFI_MANAGED_NETWORK_PROTOCOL.GetModeData()......................................1133
    EFI_MANAGED_NETWORK_PROTOCOL.Configure()..........................................1135
    EFI_MANAGED_NETWORK_PROTOCOL.McastIpToMac()..................................1137
    EFI_MANAGED_NETWORK_PROTOCOL.Groups().............................................1138
    EFI_MANAGED_NETWORK_PROTOCOL.Transmit()............................................1139
    EFI_MANAGED_NETWORK_PROTOCOL.Receive().............................................1144
    EFI_MANAGED_NETWORK_PROTOCOL.Cancel().............................................1145
    EFI_MANAGED_NETWORK_PROTOCOL.Poll()..................................................1146

26 Network Protocols — Bluetooth ......................................................................1147
  26.1 EFI Bluetooth Host Controller Protocol..................................................1147
    EFI_BLUETOOTH_HC_PROTOCOL....................................................................1147
    BLUETOOTH_HC_PROTOCOL.SendCommand()..............................................1148
<table>
<thead>
<tr>
<th>Function/Method</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>BLUETOOTH_HC_PROTOCOL.ReceiveEvent()</td>
<td>1149</td>
</tr>
<tr>
<td>BLUETOOTH_HC_PROTOCOL.AsyncReceiveEvent()</td>
<td>1150</td>
</tr>
<tr>
<td>BLUETOOTH_HC_PROTOCOL.SendACLData()</td>
<td>1151</td>
</tr>
<tr>
<td>BLUETOOTH_HC_PROTOCOL.ReceiveACLData()</td>
<td>1152</td>
</tr>
<tr>
<td>BLUETOOTH_HC_PROTOCOL.AsyncReceiveACLData()</td>
<td>1153</td>
</tr>
<tr>
<td>BLUETOOTH_HC_PROTOCOL.SendSCOData()</td>
<td>1154</td>
</tr>
<tr>
<td>BLUETOOTH_HC_PROTOCOL.ReceiveSCOData()</td>
<td>1155</td>
</tr>
<tr>
<td>BLUETOOTH_HC_PROTOCOL.AsyncReceiveSCOData()</td>
<td>1156</td>
</tr>
</tbody>
</table>

26.2 EFI Bluetooth Bus Protocol ......................................................... 1157

<table>
<thead>
<tr>
<th>Protocol</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>EFI_BLUETOOTH_IO_SERVICE_BINDING_PROTOCOL</td>
<td>1157</td>
</tr>
<tr>
<td>EFI_BLUETOOTH_IO_PROTOCOL</td>
<td>1157</td>
</tr>
<tr>
<td>BLUETOOTH_IO_PROTOCOL.GetDeviceInfo</td>
<td>1159</td>
</tr>
<tr>
<td>BLUETOOTH_IO_PROTOCOL.GetSdpInfo</td>
<td>1160</td>
</tr>
<tr>
<td>BLUETOOTH_IO_PROTOCOL.L2CapRawSend</td>
<td>1161</td>
</tr>
<tr>
<td>BLUETOOTH_IO_PROTOCOL.L2CapRawReceive</td>
<td>1162</td>
</tr>
<tr>
<td>BLUETOOTH_IO_PROTOCOL.L2CapRawAsyncReceive</td>
<td>1163</td>
</tr>
<tr>
<td>BLUETOOTH_IO_PROTOCOL.L2CapSend</td>
<td>1164</td>
</tr>
<tr>
<td>BLUETOOTH_IO_PROTOCOL.L2CapReceive</td>
<td>1165</td>
</tr>
<tr>
<td>BLUETOOTH_IO_PROTOCOL.L2CapAsyncReceive</td>
<td>1166</td>
</tr>
<tr>
<td>BLUETOOTH_IO_PROTOCOL.L2CapConnect</td>
<td>1167</td>
</tr>
<tr>
<td>BLUETOOTH_IO_PROTOCOL.L2CapDisconnect</td>
<td>1168</td>
</tr>
<tr>
<td>BLUETOOTH_IO_PROTOCOL.L2CapRegisterService</td>
<td>1169</td>
</tr>
</tbody>
</table>

26.3 EFI Bluetooth Configuration Protocol ........................................ 1170

<table>
<thead>
<tr>
<th>Protocol</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>EFI_BLUETOOTH_CONFIG_PROTOCOL</td>
<td>1170</td>
</tr>
<tr>
<td>BLUETOOTH_CONFIG_PROTOCOL.Init</td>
<td>1171</td>
</tr>
<tr>
<td>BLUETOOTH_CONFIG_PROTOCOL.Scan</td>
<td>1172</td>
</tr>
<tr>
<td>BLUETOOTH_CONFIG_PROTOCOL.Connect</td>
<td>1173</td>
</tr>
<tr>
<td>BLUETOOTH_CONFIG_PROTOCOL.Disconnect</td>
<td>1174</td>
</tr>
<tr>
<td>BLUETOOTH_CONFIG_PROTOCOL.GetData</td>
<td>1175</td>
</tr>
<tr>
<td>BLUETOOTH_CONFIG_PROTOCOL.SetData</td>
<td>1177</td>
</tr>
<tr>
<td>BLUETOOTH_CONFIG_PROTOCOL.GetRemoteData</td>
<td>1178</td>
</tr>
<tr>
<td>BLUETOOTH_CONFIG_PROTOCOL.RegisterPinCallback</td>
<td>1179</td>
</tr>
<tr>
<td>BLUETOOTH_CONFIG_PROTOCOL.RegisterGetLinkKeyCallback</td>
<td>1181</td>
</tr>
<tr>
<td>BLUETOOTH_CONFIG_PROTOCOL.RegisterSetLinkKeyCallback</td>
<td>1182</td>
</tr>
<tr>
<td>BLUETOOTH_CONFIG_PROTOCOL.RegisterLinkConnectCompleteCallback</td>
<td>1183</td>
</tr>
</tbody>
</table>

26.4 EFI Bluetooth Attribute Protocol .............................................. 1185

<table>
<thead>
<tr>
<th>Protocol</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>EFI_BLUETOOTH_ATTRIBUTE_PROTOCOL</td>
<td>1185</td>
</tr>
<tr>
<td>EFI_BLUETOOTH_ATTRIBUTE_SERVICE_BINDING_PROTOCOL</td>
<td>1194</td>
</tr>
</tbody>
</table>

26.5 EFI Bluetooth LE Configuration Protocol ...................................... 1194

<table>
<thead>
<tr>
<th>Protocol</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>EFI_BLUETOOTH_LE_CONFIG_PROTOCOL</td>
<td>1194</td>
</tr>
<tr>
<td>BLUETOOTH_LE_CONFIG_PROTOCOL.Init</td>
<td>1196</td>
</tr>
<tr>
<td>BLUETOOTH_LE_CONFIG_PROTOCOL.Scan</td>
<td>1196</td>
</tr>
<tr>
<td>BLUETOOTH_LE_CONFIG_PROTOCOL.Connect</td>
<td>1198</td>
</tr>
<tr>
<td>BLUETOOTH_LE_CONFIG_PROTOCOL.Disconnect</td>
<td>1200</td>
</tr>
<tr>
<td>BLUETOOTH_LE_CONFIG_PROTOCOL.GetData</td>
<td>1201</td>
</tr>
<tr>
<td>BLUETOOTH_LE_CONFIG_PROTOCOL.SetData</td>
<td>1202</td>
</tr>
<tr>
<td>BLUETOOTH_LE_CONFIG_PROTOCOL.GetRemoteData</td>
<td>1202</td>
</tr>
</tbody>
</table>
27 Network Protocols — VLAN, EAP, Wi-Fi and Supplicant ................................. 1214

27.1 VLAN Configuration Protocol.................................................................................................... 1214

EFI_VLAN_CONFIG_PROTOCOL.......................................................................................... 1214
- EFI_VLAN_CONFIG_PROTOCOL.Set() ........................................................................... 1214
- EFI_VLAN_CONFIG_PROTOCOL.Find() .......................................................................... 1215
- EFI_VLAN_CONFIG_PROTOCOL.Remove () ...................................................................... 1217

27.2 EAP Protocol ......................................................................................................................... 1217

EFI_EAP_PROTOCOL..................................................................................................... 1217
- EFI_EAP.SetDesiredAuthMethod()................................................................................ 1218
- EFI_EAP.RegisterAuthMethod() .................................................................................. 1219

27.2.1 EAP Management Protocol ......................................................................................... 1221

EFI_EAP_MANAGEMENT_PROTOCOL....................................................................... 1221
- EFI_EAP_MANAGEMENT.GetSystemConfiguration() .................................................... 1222
- EFI_EAP_MANAGEMENT.SetSystemConfiguration() .................................................... 1223
- EFI_EAP_MANAGEMENT.InitializePort() ...................................................................... 1224
- EFI_EAP_MANAGEMENT.UserLogon() ......................................................................... 1224
- EFI_EAP_MANAGEMENT.UserLogoff() ........................................................................... 1225
- EFI_EAP_MANAGEMENT.GetSupplicantStatus() ........................................................... 1225
- EFI_EAP_MANAGEMENT.SetSupplicantConfiguration() ............................................... 1228
- EFI_EAP_MANAGEMENT.GetSupplicantStatistics() ...................................................... 1228

27.2.2 EFI EAP Management2 Protocol .................................................................................. 1230

EFI_EAP_MANAGEMENT2_PROTOCOL......................................................................... 1230
- EFI_EAP_MANAGEMENT2_PROTOCOL.GetKey() .......................................................... 1231

27.2.3 EFI EAP Configuration Protocol ................................................................................. 1232

EFI_EAP_CONFIGURATION_PROTOCOL . ....................................................................... 1232
- EFI_EAP_CONFIGURATION_PROTOCOL.SetData()........................................................ 1232
- EFI_EAP_CONFIGURATION_PROTOCOL.GetData() ....................................................... 1235

27.3 EFI Wireless MAC Connection Protocol ................................................................. 1236

EFI_WIRELESS_MAC_CONNECTION_PROTOCOL ...................................................... 1236
- EFI_WIRELESS_MAC_CONNECTION_PROTOCOL.Scan() ............................................ 1237
- EFI_WIRELESS_MAC_CONNECTION_PROTOCOL.Associate() .................................... 1247
- EFI_WIRELESS_MAC_CONNECTION_PROTOCOL.Disassociate() ................................ 1252
- EFI_WIRELESS_MAC_CONNECTION_PROTOCOL.Authenticate() ................................ 1254
- EFI_WIRELESS_MAC_CONNECTION_PROTOCOL.Deauthenticate() .............................. 1258

27.4 EFI Wireless MAC Connection II Protocol ............................................................. 1259

EFI_WIRELESS_MAC_CONNECTION_II_PROTOCOL ................................................. 1259
- EFI_WIRELESS_MAC_CONNECTION_II_PROTOCOL.GetNetworks() ............................ 1260
- EFI_WIRELESS_MAC_CONNECTION_II_PROTOCOL.ConnectNetwork() ...................... 1264
- EFI_WIRELESS_MAC_CONNECTION_II_PROTOCOL.DisconnectNetwork() ............... 1268

27.5 EFI Supplicant Protocol ................................................................................................. 1269

27.5.1 Supplicant Service Binding Protocol ........................................................................ 1269
28 Network Protocols — TCP, IP, IPsec, FTP, TLS and Configurations .......... 1280

28.1 EFI TCPv4 Protocol................................................................. 1280
   28.1.1 TCP4 Service Binding Protocol ....................................... 1280
      EFI_TCP4_SERVICE_BINDING_PROTOCOL................................. 1280

28.1.2 TCP4 Protocol................................................................. 1280
   EFI_TCP4_PROTOCOL ................................................................ 1280
   EFI_TCP4_PROTOCOL.GetModeData().................................... 1282
   EFI_TCP4_PROTOCOL.Configure()....................................... 1286
   EFI_TCP4_PROTOCOL.Routes()........................................ 1288
   EFI_TCP4_PROTOCOL.Connect()....................................... 1289
   EFI_TCP4_PROTOCOL.Accept()....................................... 1292
   EFI_TCP4_PROTOCOL.Transmit().................................... 1293
   EFI_TCP4_PROTOCOL.Receive().................................. 1297
   EFI_TCP4_PROTOCOL.Close().................................... 1299
   EFI_TCP4_PROTOCOL.Poll()........................................ 1301

28.2 EFI TCPv6 Protocol................................................................. 1302
   28.2.1 TCPv6 Service Binding Protocol ..................................... 1302
      EFI_TCP6_SERVICE_BINDING_PROTOCOL............................... 1302

28.2.2 TCPv6 Protocol................................................................. 1303
   EFI_TCP6_PROTOCOL ................................................................ 1303
   EFI_TCP6_PROTOCOL.GetModeData()................................ 1304
   EFI_TCP6_PROTOCOL.Configure().................................. 1309
   EFI_TCP6_PROTOCOL.Connect().................................. 1310
   EFI_TCP6_PROTOCOL.Accept()................................ 1312
   EFI_TCP6_PROTOCOL.Transmit().................................. 1314
   EFI_TCP6_PROTOCOL.Receive().................................. 1318
   EFI_TCP6_PROTOCOL.Close().................................. 1320
   EFI_TCP6_PROTOCOL.Cancel().................................. 1321
   EFI_TCP6_PROTOCOL.Poll().................................... 1322

28.3 EFI IPv4 Protocol................................................................. 1323
   28.3.1 IP4 Service Binding Protocol ....................................... 1323
      EFI_IP4_SERVICE_BINDING_PROTOCOL................................. 1323

28.3.2 IP4 Protocol................................................................. 1324
   EFI_IP4_PROTOCOL ................................................................ 1324
   EFI_IP4_PROTOCOL.GetModeData()................................ 1325
   EFI_IP4_PROTOCOL.Configure().................................. 1325
   EFI_IP4_PROTOCOL.Groups().................................... 1330
   EFI_IP4_PROTOCOL.Routes().................................... 1332
### 28.8.4 EFI IPsec2 Protocol

- EFI_IPSEC2_PROTOCOL.ProcessExt() .........................................................1411
- EFI_IPSEC2_PROTOCOL.ProcessExt() .........................................................1411

### 28.9 Network Protocol - EFI FTP Protocol

- EFI_FTP4_SERVICE_BINDING_PROTOCOL Summary ..................................1413
- EFI_FTP4_PROTOCOL ..................................................................................1414
- EFI_FTP4_PROTOCOL.GetModeData() .......................................................1415
- EFI_FTP4_PROTOCOL.Connect() ................................................................1415
- EFI_FTP4_PROTOCOL.Close() ...................................................................1417
- EFI_FTP4_PROTOCOL.Configure() ..............................................................1418
- EFI_FTP4_PROTOCOL.ReadFile() .............................................................1420
- EFI_FTP4_PROTOCOL.WriteFile() ............................................................1423
- EFI_FTP4_PROTOCOL.ReadDirectory() ......................................................1424
- EFI_FTP4_PROTOCOL.Poll() ....................................................................1425

### 28.10 EFI TLS Protocols

- EFI_TLS_SERVICE_BINDING_PROTOCOL ..................................................1426
- EFI_TLS_PROTOCOL ................................................................................1426
- EFI_TLS_PROTOCOL.SetSessionData() .....................................................1427
- EFI_TLS_PROTOCOL.GetSessionData() ....................................................1434
- EFI_TLS_PROTOCOL.BuildResponsePacket() ...........................................1435
- EFI_TLS_PROTOCOL.ProcessPacket() .......................................................1436

### 28.10.3 EFI TLS Configuration Protocol

- EFI_TLS_CONFIGURATION_PROTOCOL ....................................................1438
- EFI_TLS_CONFIGURATION_PROTOCOL.SetData() ......................................1439
- EFI_TLS_CONFIGURATION_PROTOCOL.GetData() .....................................1440

### 29 Network Protocols — ARP, DHCP, DNS, HTTP and REST

#### 29.1 ARP Protocol

- EFI_ARP_SERVICE_BINDING_PROTOCOL ..................................................1442
- EFI_ARP_PROTOCOL ..............................................................................1443
- EFI_ARP_PROTOCOL.SetSessionData() ....................................................1444
- EFI_ARP_PROTOCOL.GetSessionData() ....................................................1446
- EFI_ARP_PROTOCOL.Find() ....................................................................1448
- Related Definitions ................................................................................1449
- EFI_ARP_PROTOCOL.Delete() .................................................................1449
- EFI_ARP_PROTOCOL.Flush() ..................................................................1450
- EFI_ARP_PROTOCOL.Request() ...............................................................1451
- EFI_ARP_PROTOCOL.Cancel() ...............................................................1452

#### 29.2 EFI DHCPv4 Protocol

- EFI_DHCP4_SERVICE_BINDING_PROTOCOL ............................................1453
- EFI_DHCP4_PROTOCOL ..........................................................................1454
- EFI_DHCP4_PROTOCOL.SetModeData() ....................................................1455
- EFI_DHCP4_PROTOCOL.Configure() .......................................................1458
- EFI_DHCP4_PROTOCOL.Start() .............................................................1465
29.3 EFI DHCP6 Protocol ........................................................................................................1474
  29.3.1 DHCP6 Service Binding Protocol ........................................................................1474
    EFI_DHCP6_SERVICE_BINDING_PROTOCOL ........................................................................1474
  29.3.2 DHCP6 Protocol ..................................................................................................1474
    EFI_DHCP6_PROTOCOL .................................................................................................1474
    EFI_DHCP6_PROTOCOL.Cancel() ..................................................................................1476
    EFI_DHCP6_PROTOCOL.Poll() .......................................................................................1481
    EFI_DHCP6_PROTOCOL.Start() .......................................................................................1487
    EFI_DHCP6_PROTOCOL.InfoRequest() ............................................................................1488
    EFI_DHCP6_PROTOCOL.RenewRebind() .........................................................................1491
    EFI_DHCP6_PROTOCOL.Decline() ...................................................................................1492
    EFI_DHCP6_PROTOCOL.Release() ...................................................................................1494
    EFI_DHCP6_PROTOCOL.Stop() .......................................................................................1495
    EFI_DHCP6_PROTOCOL.Parse() ......................................................................................1496

29.4 EFI DNSv4 Protocol ..................................................................................................1497
  29.4.1 DNS Service Binding Protocol ...........................................................................1497
    EFI_DNS4_SERVICE_BINDING_PROTOCOL ........................................................................1497
  29.4.2 DNS Protocol ....................................................................................................1498
    EFI_DNS4_PROTOCOL ....................................................................................................1498
    EFI_DNS4_PROTOCOL.GetModeData() ............................................................................1499
    EFI_DNS4_PROTOCOL.Configure() ................................................................................1501
    EFI_DNS4_PROTOCOLHostNameToIp() ..........................................................................1502
    EFI_DNS4_PROTOCOL.IpToHostName() ..........................................................................1505
    EFI_DNS4_PROTOCOL.GeneralLookUp() .......................................................................1506
    EFI_DNS4_PROTOCOL.UpdateDnsCache() .....................................................................1507
    EFI_DNS4_PROTOCOL.Poll() .........................................................................................1508
    EFI_DNS4_PROTOCOL.Cancel() .....................................................................................1509

29.5 EFI DNSv6 Protocol ..................................................................................................1510
  29.5.1 DNS6 Service Binding Protocol .........................................................................1510
    EFI_DNS6_SERVICE_BINDING_PROTOCOL ........................................................................1510
  29.5.2 DNS6 Protocol ..................................................................................................1511
    EFI_DNS6_PROTOCOL ....................................................................................................1511
    EFI_DNS6_PROTOCOL.GetModeData() ............................................................................1512
    EFI_DNS6_PROTOCOL.Configure() ................................................................................1515
    EFI_DNS6_PROTOCOL.HostNameToIp() ..........................................................................1516
    EFI_DNS6_PROTOCOL.IpToHostName() ..........................................................................1520
    EFI_DNS6_PROTOCOL.GeneralLookUp() .......................................................................1520
    EFI_DNS6_PROTOCOL.UpdateDnsCache() .....................................................................1522
    EFI_DNS6_PROTOCOL.Poll() .........................................................................................1522
    EFI_DNS6_PROTOCOL.Cancel() .....................................................................................1523

29.6 EFI HTTP Protocols ...............................................................................................1524
  29.6.1 HTTP Service Binding Protocol .......................................................................1524
    EFI_HTTP_SERVICE_BINDING_PROTOCOL ........................................................................1524
## Table of Contents

<table>
<thead>
<tr>
<th>Section</th>
<th>Title</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>29.6.2</td>
<td>EFI HTTP Protocol Specific Definitions</td>
<td>1525</td>
</tr>
<tr>
<td></td>
<td>EFI_HTTP_PROTOCOL</td>
<td>1525</td>
</tr>
<tr>
<td></td>
<td>EFI_HTTP_PROTOCOL.GetModeData()</td>
<td>1526</td>
</tr>
<tr>
<td></td>
<td>EFI_HTTP_PROTOCOL.Configure()</td>
<td>1528</td>
</tr>
<tr>
<td></td>
<td>EFI_HTTP_PROTOCOL.Request()</td>
<td>1529</td>
</tr>
<tr>
<td></td>
<td>EFI_HTTP_PROTOCOL.Cancel()</td>
<td>1534</td>
</tr>
<tr>
<td></td>
<td>EFI_HTTP_PROTOCOL.Response()</td>
<td>1535</td>
</tr>
<tr>
<td></td>
<td>EFI_HTTP_PROTOCOL.Poll()</td>
<td>1536</td>
</tr>
<tr>
<td>29.6.3</td>
<td>HTTP Utilities Protocol</td>
<td>1543</td>
</tr>
<tr>
<td></td>
<td>EFI_HTTP_UTILITIES_PROTOCOL</td>
<td>1543</td>
</tr>
<tr>
<td></td>
<td>EFI_HTTP_UTILITIES_PROTOCOL.Build()</td>
<td>1543</td>
</tr>
<tr>
<td></td>
<td>EFI_HTTP_UTILITIES_PROTOCOL.Parse()</td>
<td>1545</td>
</tr>
<tr>
<td>29.7</td>
<td>EFI REST Support Overview</td>
<td>1546</td>
</tr>
<tr>
<td>29.7.1</td>
<td>EFI REST Protocol</td>
<td>1551</td>
</tr>
<tr>
<td></td>
<td>EFI_REST_PROTOCOL</td>
<td>1551</td>
</tr>
<tr>
<td></td>
<td>EFI_REST_PROTOCOL.SendReceive()</td>
<td>1551</td>
</tr>
<tr>
<td></td>
<td>EFI_REST_PROTOCOL.GetServiceTime()</td>
<td>1553</td>
</tr>
<tr>
<td>29.7.2</td>
<td>EFI REST EX Protocol</td>
<td>1553</td>
</tr>
<tr>
<td></td>
<td>EFI_REST_EX_PROTOCOL</td>
<td>1553</td>
</tr>
<tr>
<td></td>
<td>EFI_REST_EX_PROTOCOL.SendReceive()</td>
<td>1554</td>
</tr>
<tr>
<td></td>
<td>EFI_REST_EX_PROTOCOL.GetService()</td>
<td>1555</td>
</tr>
<tr>
<td></td>
<td>EFI_REST_EX_PROTOCOL.GetModeData()</td>
<td>1556</td>
</tr>
<tr>
<td></td>
<td>EFI_REST_EX_PROTOCOL.Configure()</td>
<td>1557</td>
</tr>
<tr>
<td></td>
<td>EFI_REST_EX_PROTOCOL.AsyncSendReceive()</td>
<td>1559</td>
</tr>
<tr>
<td></td>
<td>EFI_REST_EX_PROTOCOL.EventService()</td>
<td>1560</td>
</tr>
<tr>
<td>29.7.3</td>
<td>EFI REST JSON Resource to C Structure Converter</td>
<td>1576</td>
</tr>
<tr>
<td></td>
<td>EFI_REST_JSON_STRUCTURE.Register ()</td>
<td>1577</td>
</tr>
<tr>
<td></td>
<td>EFI_REST_JSON_STRUCTURE.ToStructure ()</td>
<td>1579</td>
</tr>
<tr>
<td></td>
<td>EFI_REST_JSON_STRUCTURE.ToJson ()</td>
<td>1580</td>
</tr>
<tr>
<td></td>
<td>EFI_REST_JSON_STRUCTURE.DestroyStructure ()</td>
<td>1583</td>
</tr>
<tr>
<td>30</td>
<td>Network Protocols — UDP and MTFTP</td>
<td>1584</td>
</tr>
<tr>
<td>30.1</td>
<td>EFI UDP Protocol</td>
<td>1584</td>
</tr>
<tr>
<td>30.1.1</td>
<td>UDP4 Service Binding Protocol</td>
<td>1584</td>
</tr>
<tr>
<td></td>
<td>EFI_UDP4_SERVICE_BINDING_PROTOCOL</td>
<td>1584</td>
</tr>
<tr>
<td>30.1.2</td>
<td>UDP4 Protocol</td>
<td>1584</td>
</tr>
<tr>
<td></td>
<td>EFI_UDP4_PROTOCOL</td>
<td>1584</td>
</tr>
<tr>
<td></td>
<td>EFI_UDP4_PROTOCOL.GetModeData()</td>
<td>1586</td>
</tr>
<tr>
<td></td>
<td>EFI_UDP4_PROTOCOL.Configure()</td>
<td>1588</td>
</tr>
<tr>
<td></td>
<td>EFI_UDP4_PROTOCOL.Groups()</td>
<td>1590</td>
</tr>
<tr>
<td></td>
<td>EFI_UDP4_PROTOCOL.Routes()</td>
<td>1591</td>
</tr>
<tr>
<td></td>
<td>EFI_UDP4_PROTOCOL.Transmit()</td>
<td>1593</td>
</tr>
<tr>
<td></td>
<td>EFI_UDP4_PROTOCOL.Receive()</td>
<td>1598</td>
</tr>
<tr>
<td></td>
<td>EFI_UDP4_PROTOCOL.Cancel()</td>
<td>1600</td>
</tr>
<tr>
<td></td>
<td>EFI_UDP4_PROTOCOL.Poll()</td>
<td>1600</td>
</tr>
</tbody>
</table>
# 30.2 EFI UDPv6 Protocol

- **EFI_UDP6_SERVICE_BINDING_PROTOCOL**
- EFI_UDP6_PROTOCOL
  - EFI_UDP6_PROTOCOL.GetModeData()
  - EFI_UDP6_PROTOCOL.Configure()
  - EFI_UDP6_PROTOCOL.Groups()
  - EFI_UDP6_PROTOCOL.Transmit()
  - EFI_UDP6_PROTOCOL.Receive()
  - EFI_UDP6_PROTOCOL.Cancel()
  - EFI_UDP6_PROTOCOL.Poll()

# 30.3 EFI MTFTPv4 Protocol

- **EFI_MTFTP4_SERVICE_BINDING_PROTOCOL**
- EFI_MTFTP4_PROTOCOL
  - EFI_MTFTP4_PROTOCOL.GetModeData()
  - EFI_MTFTP4_PROTOCOL.Configure()
  - EFI_MTFTP4_PROTOCOL.GetInfo()
  - EFI_MTFTP4_PROTOCOL.ParseOptions()
  - EFI_MTFTP4_PROTOCOL.ReadFile()
  - EFI_MTFTP4_PROTOCOL.WriteFile()
  - EFI_MTFTP4_PROTOCOL.ReadDirectory()
  - EFI_MTFTP4_PROTOCOL.POLL()

# 30.4 EFI MTFTPv6 Protocol

- **EFI_MTFTP6_SERVICE_BINDING_PROTOCOL**
- EFI_MTFTP6_PROTOCOL
  - EFI_MTFTP6_PROTOCOL.GetModeData()
  - EFI_MTFTP6_PROTOCOL.Configure()
  - EFI_MTFTP6_PROTOCOL.GetInfo()
  - EFI_MTFTP6_PROTOCOL.ParseOptions()
  - EFI_MTFTP6_PROTOCOL.ReadFile()
  - EFI_MTFTP6_PROTOCOL.WriteByte()
  - EFI_MTFTP6_PROTOCOL.ReadDirectory()
  - EFI_MTFTP6_PROTOCOL.POLL()

# 31 EFI Redfish Service Support

- **EFI_REDFISH_DISCOVER_PROTOCOL**
  - EFI_REDFISH_DISCOVER_PROTOCOL.GetNetworkInterfaceList()
  - EFI_REDFISH_DISCOVER_PROTOCOL.AcquireRedfishService()
  - EFI_REDFISH_DISCOVER_PROTOCOL.AbortAcquireRedfishService()
  - EFI_REDFISH_DISCOVER_PROTOCOL.ReleaseRedfishService()
33 Human Interface Infrastructure Overview ......................................................... 1717

33.1 Goals ........................................................................................................... 1717

33.2 Design Discussion ...................................................................................... 1718

33.2.1 Drivers And Applications ................................................................. 1718

33.2.2 Localization .......................................................................................... 1726

33.2.3 User Input .............................................................................................. 1727

33.2.4 Keyboard Layout ................................................................................. 1728

33.2.5 Forms ...................................................................................................... 1731

33.2.6 Strings .................................................................................................... 1760

33.2.7 Fonts ....................................................................................................... 1764

33.2.8 Images ................................................................................................... 1770

33.2.9 HII Database ........................................................................................ 1771

32 Secure Boot and Driver Signing ...................................................................... 1684

32.1 Secure Boot .............................................................................................. 1684

32.2 UEFI Driver Signing Overview .................................................................. 1689

32.2.1 Digital Signatures ............................................................................... 1689

32.2.2 Embedded Signatures ......................................................................... 1691

32.2.3 Creating Image Digests from Images .................................................. 1692

32.2.4 Code Definitions ................................................................................... 1692

32.2.5 UEFI Image Validation ........................................................................ 1709

32.2.6 Platform Firmware Key Storage Requirements .................................... 1700

32.2.7 Fonts ....................................................................................................... 1764

32.2.8 Images ................................................................................................... 1770

32.2.9 HII Database ........................................................................................ 1771
33.2.10 Forms Browser ........................................................................................................... 1772
33.2.11 Configuration Settings .............................................................................................. 1776
33.2.12 Form Callback Logic .................................................................................................. 1779
33.2.13 Driver Model Interaction ......................................................................................... 1782
33.2.14 Human Interface Component Interactions ............................................................... 1783
33.2.15 Standards Map Forms ............................................................................................. 1784
33.3 Code Definitions ............................................................................................................ 1788
   33.3.1 Package Lists and Package Headers ........................................................................... 1789
      EFI_HII_PACKAGE_HEADER ............................................................................................ 1789
   33.3.2 Simplified Font Package .......................................................................................... 1791
33.3.3 Font Package ............................................................................................................. 1794
33.3.4 Device Path Package .................................................................................................. 1806
33.3.5 GUID Package .......................................................................................................... 1807
33.3.6 String Package ......................................................................................................... 1807
33.3.7 Image Package .......................................................................................................... 1823
33.3.8 Forms Package ......................................................................................................... 1840
33.3.9 Keyboard Package .................................................................................................... 1917
33.3.10 Animations Package ............................................................................................... 1917
34 HII Protocols .................................................................................................................. 1929
34.1 Font Protocol ................................................................................................................ 1929
   EFI_HII_FONT_PROTOCOL .................................................................................... 1929
   EFI_HII_FONT_PROTOCOL.GetFontInfo() ...................................................................... 1932
   EFI_HII_FONT_PROTOCOL.GetGlyph() ......................................................................... 1935
   EFI_HII_FONT_PROTOCOL.GetStringIdToImage() ....................................................... 1933
   EFI_HII_FONT_PROTOCOL.StringToImage() ............................................................... 1929
   EFI_HII_FONT_PROTOCOL....................................................................................... 1929
34.2 EFI HII Font Ex Protocol ............................................................................................. 1937
   EFI_HII_FONT_EX_PROTOCOL .................................................................................... 1937
   EFI_HII_FONT_EX_PROTOCOL.GetFontInfoEx() ......................................................... 1942
   EFI_HII_FONT_EX_PROTOCOL.GetGlyphInfo() .......................................................... 1942
   EFI_HII_FONT_EX_PROTOCOL.GetGlyphEx() ............................................................. 1940
   EFI_HII_FONT_EX_PROTOCOL.GetStringIdToImageEx() .............................................. 1939
   EFI_HII_FONT_EX_PROTOCOL.StringToImageEx() ...................................................... 1938
34.2.1 Code Definitions ....................................................................................................... 1943
   EFI_FONT_DISPLAY_INFO ......................................................................................... 1943
   EFI_IMAGE_OUTPUT ............................................................................................... 1945
34.3 String Protocol ............................................................................................................. 1946
   EFI_HII_STRING_PROTOCOL .................................................................................... 1946
   EFI_HII_STRING_PROTOCOL.GetLanguages() .............................................................. 1951
   EFI_HII_STRING_PROTOCOL.GetSecondaryLanguages() ............................................ 1952
   EFI_HII_STRING_PROTOCOL.GetString() .................................................................... 1950
   EFI_HII_STRING_PROTOCOL.SetString() ..................................................................... 1950
   EFI_HII_STRING_PROTOCOL.NewString() ................................................................... 1946
34.4 Image Protocol ............................................................................................................. 1953
   EFI_HII_IMAGE_PROTOCOL .................................................................................... 1953
   EFI_HII_IMAGE_PROTOCOL.GetImage() ...................................................................... 1955
   EFI_HII_IMAGE_PROTOCOL.NewImage() ...................................................................... 1954
34.5 EFI HII Image Ex Protocol ................................................................. 1960
  EFI_HII_IMAGE_EX_PROTOCOL ................................................................. 1961
  EFI_HII_IMAGE_EX_PROTOCOL.NewImageEx() .............................................. 1961
  EFI_HII_IMAGE_EX_PROTOCOL.GetImageEx() .............................................. 1962
  EFI_HII_IMAGE_EX_PROTOCOL.SetImageEx() .............................................. 1963
  EFI_HII_IMAGE_EX_PROTOCOL.DrawImageEx() ........................................... 1963
  EFI_HII_IMAGE_EX_PROTOCOL.DrawImageIdEx() .......................................... 1964
  EFI_HII_IMAGE_EX_PROTOCOL.GetImageInfo() ........................................... 1965

34.6 EFI HII Image Decoder Protocol .......................................................... 1966
  EFI_HII_IMAGE_DECODER_PROTOCOL.DecodeImage() .................................... 1967
  EFI_HII_IMAGE_DECODER_PROTOCOL.GetImageDecoderName() .................... 1968
  EFI_HII_IMAGE_DECODER_PROTOCOL.GetImageInfo() .................................... 1969
  EFI_HII_IMAGE_DECODER_PROTOCOL.Decode() ........................................... 1972

34.7 Font Glyph Generator Protocol .......................................................... 1973
  EFI_HII_FONT_GLYPH_GENERATOR_PROTOCOL ........................................... 1973
  EFI_HII_FONT_GLYPH_GENERATOR_PROTOCOL.GenerateGlyph() .................. 1973
  EFI_HII_FONT_GLYPH_GENERATOR_PROTOCOL.GenerateGlyphImage() .......... 1974

34.8 Database Protocol ............................................................................. 1976
  EFI_HII_DATABASE_PROTOCOL ................................................................. 1976
  EFI_HII_DATABASE_PROTOCOL.NewPackageList() ......................................... 1977
  EFI_HII_DATABASE_PROTOCOL.RemovePackageList() .................................... 1978
  EFI_HII_DATABASE_PROTOCOL.UpdatePackageList() .................................... 1979
  EFI_HII_DATABASE_PROTOCOL.ListPackageLists() ....................................... 1980
  EFI_HII_DATABASE_PROTOCOL.ExportPackageLists() ................................... 1981
  EFI_HII_DATABASE_PROTOCOL.RegisterPackageNotify() ............................ 1982
  EFI_HII_DATABASE_PROTOCOL.UnregisterPackageNotify() ......................... 1984
  EFI_HII_DATABASE_PROTOCOL.FindKeyboardLayouts() .............................. 1984
  EFI_HII_DATABASE_PROTOCOL.GetKeyboardLayout() ................................... 1985
  EFI_HII_DATABASE_PROTOCOL.SetKeyboardLayout() ................................... 1991
  EFI_HII_DATABASE_PROTOCOL.GetPackageListHandle() ................................ 1992

34.8.1 Database Structures ....................................................................... 1993
  EFI_HII_DATABASE_NOTIFY ................................................................. 1993
  EFI_HII_DATABASE_NOTIFY_TYPE ......................................................... 1994

35 HII Configuration Processing and Browser Protocol ....................... 1995

  35.1 Introduction ...................................................................................... 1995
    35.1.1 Common Configuration Data Format ........................................... 1995
    35.1.2 Data Flow .................................................................................. 1995
  35.2 Configuration Strings ....................................................................... 1995
    35.2.1 String Syntax ............................................................................. 1995
    35.2.2 String Types ............................................................................. 2001
  35.3 EFI Configuration Keyword Handler Protocol .................................. 2002
    EFI_CONFIG_KEYWORD_HANDLER_PROTOCOL .................................. 2002
    EFI.Keyword_HANDLER_PROTOCOL.SetData() ....................................... 2003
36.3.2 Credential Provider Protocols ...................................................................................... 2056
  EFI_USER_CREDENTIAL2_PROTOCOL ............................................................................ 2056
  EFI_USER_CREDENTIAL2_PROTOCOL.Enroll() ............................................................... 2058
  EFI_USER_CREDENTIAL2_PROTOCOL.Form() ................................................................ 2059
  EFI_USER_CREDENTIAL2_PROTOCOL.Title() .................................................................. 2060
  EFI_USER_CREDENTIAL2_PROTOCOL.Title() .................................................................. 2061
  EFI_USER_CREDENTIAL2_PROTOCOL.User() .................................................................. 2061
  EFI_USER_CREDENTIAL2_PROTOCOL.Select() .................................................................. 2063
  EFI_USER_CREDENTIAL2_PROTOCOL.Deselect() ............................................................ 2064
  EFI_USER_CREDENTIAL2_PROTOCOL.Default() .............................................................. 2064
  EFI_USER_CREDENTIAL2_PROTOCOL.GetInfo() .............................................................. 2065
  EFI_USER_CREDENTIAL2_PROTOCOL.GetNextInfo() ....................................................... 2066
  EFI_USER_CREDENTIAL2_PROTOCOL.Delete() ............................................................... 2066
36.3.3 Deferred Image Load Protocol .................................................................................... 2067
  EFI_DEFERRED_IMAGE_LOAD_PROTOCOL .................................................................. 2067
  EFI_DEFERRED_IMAGE_LOAD_PROTOCOL.GetImageInfo() ........................................... 2068
36.4 User Information ........................................................................................................... 2069
  36.4.1 EFI_USER_INFO_ACCESS_POLICY_RECORD .................................................... 2070
  36.4.2 EFI_USER_INFO_CBEFF_RECORD ...................................................................... 2074
  36.4.3 EFI_USER_INFO_CREATE_DATE_RECORD ......................................................... 2074
  36.4.4 EFI_USER_INFO_CREDENTIAL_PROVIDER_RECORD .......................................... 2075
  36.4.5 EFI_USER_INFO_CREDENTIAL_PROVIDER_NAME_RECORD .................................. 2075
  36.4.6 EFI_USER_INFO_CREDENTIAL_TYPE_RECORD .................................................... 2075
  36.4.7 EFI_USER_INFO_CREDENTIAL_TYPE_NAME_RECORD .......................................... 2076
  36.4.8 EFI_USER_INFO_GUID_RECORD .......................................................................... 2076
  36.4.9 EFI_USER_INFO_FAR_RECORD ........................................................................... 2076
  36.4.10 EFI_USER_INFO_IDENTIFIER_RECORD ............................................................ 2077
  36.4.11 EFI_USER_INFO_IDENTITY_POLICY_RECORD ................................................. 2077
  36.4.12 EFI_USER_INFO_NAME_RECORD ...................................................................... 2079
  36.4.13 EFI_USER_INFO_PKCS11_RECORD ................................................................. 2079
  36.4.14 EFI_USER_INFO_RETRY_RECORD ...................................................................... 2079
  36.4.15 EFI_USER_INFO_USAGE_DATE_RECORD ......................................................... 2080
  36.4.16 EFI_USER_INFO_USAGE_COUNT_RECORD ....................................................... 2080
36.5 User Information Table ............................................................................................... 2080
37 Secure Technologies ...................................................................................................... 2081
  37.1 Hash Overview ........................................................................................................ 2081
    37.1.1 Hash References ............................................................................................. 2081
      EFI_HASH_SERVICE_BINDING_PROTOCOL .......................................................... 2081
      EFI_HASH_PROTOCOL ......................................................................................... 2082
      EFI_HASH_PROTOCOL.GetHashSize() ..................................................................... 2082
      EFI_HASH_PROTOCOL.Hash() ............................................................................... 2083
    37.1.2 Other Code Definitions .................................................................................... 2085
      EFI_SHA1_HASH, EFI_SHA224_HASH, EFI_SHA256_HASH, EFI_SHA384_HASH,
      EFI_SHA512HASH, EFI_MD5_HASH ......................................................................... 2085
  37.2 Hash2 Protocols ....................................................................................................... 2087
    37.2.1 EFI Hash2 Service Binding Protocol .............................................................. 2087
<table>
<thead>
<tr>
<th>Section</th>
<th>Title</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>37.2.2</td>
<td>EFI Hash2 Protocol</td>
<td>2087</td>
</tr>
<tr>
<td>37.2.2</td>
<td>EFI_HASH2_PROTOCOL</td>
<td>2087</td>
</tr>
<tr>
<td>37.2.2</td>
<td>EFI_HASH2_PROTOCOL.GetHashSize()</td>
<td>2090</td>
</tr>
<tr>
<td>37.2.2</td>
<td>EFI_HASH2_PROTOCOL.Hash()</td>
<td>2091</td>
</tr>
<tr>
<td>37.2.2</td>
<td>EFI_HASH2_PROTOCOL.HashInit()</td>
<td>2092</td>
</tr>
<tr>
<td>37.2.2</td>
<td>EFI_HASH2_PROTOCOL.HashUpdate()</td>
<td>2093</td>
</tr>
<tr>
<td>37.2.2</td>
<td>EFI_HASH2_PROTOCOL.HashFinal()</td>
<td>2094</td>
</tr>
<tr>
<td>37.2.3</td>
<td>Other Code Definitions</td>
<td>2096</td>
</tr>
<tr>
<td>37.3</td>
<td>Key Management Service</td>
<td>2097</td>
</tr>
<tr>
<td>37.3</td>
<td>EFI_KEY_MANAGEMENT_SERVICE_PROTOCOL</td>
<td>2097</td>
</tr>
<tr>
<td>37.3</td>
<td>EFI_KMS_PROTOCOL.GetServiceStatus()</td>
<td>2109</td>
</tr>
<tr>
<td>37.3</td>
<td>EFI_KMS_PROTOCOL.RegisterClient()</td>
<td>2110</td>
</tr>
<tr>
<td>37.3</td>
<td>EFI_KMS_PROTOCOL.CreateKey()</td>
<td>2112</td>
</tr>
<tr>
<td>37.3</td>
<td>EFI_KMS_PROTOCOL.GetKey()</td>
<td>2114</td>
</tr>
<tr>
<td>37.3</td>
<td>EFI_KMS_PROTOCOL.AddKey()</td>
<td>2117</td>
</tr>
<tr>
<td>37.3</td>
<td>EFI_KMS_PROTOCOL.DeleteKey()</td>
<td>2119</td>
</tr>
<tr>
<td>37.3</td>
<td>EFI_KMS_PROTOCOL.GetKeyAttributes()</td>
<td>2121</td>
</tr>
<tr>
<td>37.3</td>
<td>EFI_KMS_PROTOCOL.AddKeyAttributes()</td>
<td>2123</td>
</tr>
<tr>
<td>37.3</td>
<td>EFI_KMS_PROTOCOL.DeleteKeyAttributes()</td>
<td>2126</td>
</tr>
<tr>
<td>37.3</td>
<td>EFI_KMS_PROTOCOL.GetKeyByAttributes()</td>
<td>2129</td>
</tr>
<tr>
<td>37.4</td>
<td>PKCS7 Verify Protocol</td>
<td>2131</td>
</tr>
<tr>
<td>37.4</td>
<td>EFI_PKCS7_VERIFY_PROTOCOL</td>
<td>2131</td>
</tr>
<tr>
<td>37.4</td>
<td>EFI_PKCS7_VERIFY_PROTOCOL.VerifyBuffer()</td>
<td>2133</td>
</tr>
<tr>
<td>37.4</td>
<td>EFI_PKCS7_VERIFY_PROTOCOL.VerifySignature()</td>
<td>2136</td>
</tr>
<tr>
<td>37.5</td>
<td>Random Number Generator Protocol</td>
<td>2139</td>
</tr>
<tr>
<td>37.5</td>
<td>EFI_RNG_PROTOCOL</td>
<td>2139</td>
</tr>
<tr>
<td>37.5</td>
<td>EFI_RNG_PROTOCOL.GetInfo()</td>
<td>2140</td>
</tr>
<tr>
<td>37.5</td>
<td>EFI_RNG_PROTOCOL.GetRNG()</td>
<td>2141</td>
</tr>
<tr>
<td>37.5.1</td>
<td>EFI RNG Algorithm Definitions</td>
<td>2142</td>
</tr>
<tr>
<td>37.5.2</td>
<td>RNG References</td>
<td>2143</td>
</tr>
<tr>
<td>37.6</td>
<td>Smart Card Reader and Smart Card Edge Protocols</td>
<td>2143</td>
</tr>
<tr>
<td>37.6.1</td>
<td>Smart Card Reader Protocol</td>
<td>2144</td>
</tr>
<tr>
<td>37.6.1</td>
<td>EFI_SMART_CARD_READER_PROTOCOL</td>
<td>2144</td>
</tr>
<tr>
<td>37.6.1</td>
<td>EFI_SMART_CARD_READER_PROTOCOL.SCardConnect()</td>
<td>2145</td>
</tr>
<tr>
<td>37.6.1</td>
<td>EFI_SMART_CARD_READER_PROTOCOL.SCardDisconnect()</td>
<td>2147</td>
</tr>
<tr>
<td>37.6.1</td>
<td>EFI_SMART_CARD_READER_PROTOCOL.SCardStatus()</td>
<td>2148</td>
</tr>
<tr>
<td>37.6.1</td>
<td>EFI_SMART_CARD_READER_PROTOCOL.SCardTransmit()</td>
<td>2149</td>
</tr>
<tr>
<td>37.6.1</td>
<td>EFI_SMART_CARD_READER_PROTOCOL.SCardControl()</td>
<td>2151</td>
</tr>
<tr>
<td>37.6.1</td>
<td>EFI_SMART_CARD_READER_PROTOCOL.SCardGetAttrib()</td>
<td>2152</td>
</tr>
<tr>
<td>37.6.2</td>
<td>Smart Card Edge Protocol</td>
<td>2154</td>
</tr>
<tr>
<td>37.6.2</td>
<td>EFI_SMART_CARD_EDGE_PROTOCOL</td>
<td>2154</td>
</tr>
<tr>
<td>37.6.2</td>
<td>EFI_SMART_CARD_EDGE_PROTOCOL.GetContext()</td>
<td>2156</td>
</tr>
<tr>
<td>37.6.2</td>
<td>EFI_SMART_CARD_EDGE_PROTOCOL.Connect()</td>
<td>2158</td>
</tr>
<tr>
<td>37.6.2</td>
<td>EFI_SMART_CARD_EDGE_PROTOCOL.Disconnect()</td>
<td>2159</td>
</tr>
</tbody>
</table>
38 Miscellaneous Protocols ................................................................. 2173

38.1 EFI Timestamp Protocol ................................................................. 2173
EFI_TIMESTAMP_PROTOCOL ................................................................. 2173
EFI_TIMESTAMP_PROTOCOL.GetTimestamp() ........................................... 2173
EFI_TIMESTAMP_PROTOCOL.GetProperties() .......................................... 2174

38.2 Reset Notification Protocol ........................................................... 2175
EFI_RESET_NOTIFICATION_PROTOCOL .................................................. 2175
EFI_RESET_NOTIFICATION_PROTOCOL.RegisterResetNotify() ................. 2175
EFI_RESET_NOTIFICATION_PROTOCOL.UnregisterResetNotify() ............... 2176

Appendix A GUID and Time Formats ..................................................... 2178
Appendix B Console ........................................................................... 2180
Appendix C Device Path Examples ....................................................... 2188
Appendix D Status Codes ................................................................. 2194
Appendix E Universal Network Driver Interfaces .................................. 2197
Appendix F Using the Simple Pointer Protocol .................................... 2284
Appendix G Using the EFI Extended SCSI Pass Thru Protocol .............. 2285
Appendix H Compression Source Code ................................................. 2288
Appendix I Decompression Source Code ............................................. 2318
Appendix J EFI Byte Code Virtual Machine Opcode List ....................... 2334
Appendix K Alphabetic Function Lists ............................................... 2337
Appendix L EFI 1.10 Protocol Changes and Deprecation List ............... 2338
Appendix M Formats —
Language Codes and Language Code Arrays .................................. 2341
Appendix N Common Platform Error Record ....................................... 2342
Appendix O UEFI ACPI Data Table .................................................... 2391
Appendix P Hardware Error Record Persistence Usage ....................... 2394
Appendix Q References ..................................................................... 2396
List of Tables

Table 1-1 SI prefixes .................................................................13
Table 1-2 Binary prefixes..............................................................13
Table 2-1 UEFI Image Memory Types ........................................16
Table 2-2 UEFI Runtime Services ............................................18
Table 2-3 Common UEFI Data Types .......................................20
Table 2-4 Modifiers for Common UEFI Data Types ..................21
Table 2-5 Map: EFI Cacheability Attributes to AArch64 Memory Types .................................................................38
Table 2-6 Map: UEFI Permission Attributes to ARM Paging Attributes ..........................................................38
Table 2-7 RV32 datatype alignment ......................................44
Table 2-8 RV64 datatype alignment ......................................44
Table 2-9 Register name and ABI name .................................45
Table 2-10 UEFI Protocols ..................................................47
Table 2-11 Required UEFI Implementation Elements ...........62
Table 3-1 Global Variables ................................................82
Table 3-2 UEFI Image Types .................................................89
Table 4-1 Usage of Memory Attribute Definitions ................108
Table 5-1 Legacy MBR .......................................................115
Table 5-2 Legacy MBR Partition Record ................................116
Table 5-3 Protective MBR ...................................................118
Table 5-4 Protective MBR Partition Record protecting the entire disk ..........................................................118
Table 5-5 GPT Header ..........................................................123
Table 5-6 GPT Partition Entry - Attributes .............................124
Table 5-7 Defined GPT Partition Entry - Partition Type GUIDs ........................................................................125
Table 5-8 Defined GPT Partition Entry - Attributes .................126
Table 5-9 Register name and ABI name. ................................45
Table 5-10 GPT Header ..........................................................45
Table 5-11 RV64 datatype alignment ..................................44
Table 2-7 RV32 datatype alignment ..................................44
Table 2-6 Map: UEFI Permission Attributes to ARM Paging Attributes ..................................................38
Table 2-5 Map: EFI Cacheability Attributes to AArch64 Memory Types .................................................................38
Table 2-4 Modifiers for Common UEFI Data Types ..................21
Table 2-3 Common UEFI Data Types .......................................20
Table 2-2 UEFI Runtime Services ............................................18
Table 2-1 UEFI Image Memory Types ........................................16
Table 1-2 Binary prefixes..............................................................13
Table 1-1 SI prefixes .................................................................13

UEFI Forum, Inc. March 2021
<table>
<thead>
<tr>
<th>Table Numbers</th>
<th>Table Title</th>
</tr>
</thead>
<tbody>
<tr>
<td>Table 8-8</td>
<td>Flag Firmware Behavior</td>
</tr>
<tr>
<td>Table 8-9</td>
<td>Variables Using EFI_CAPSULE_REPORT_GUID</td>
</tr>
<tr>
<td>Table 10-1</td>
<td>Generic Device Path Node Structure</td>
</tr>
<tr>
<td>Table 10-2</td>
<td>Device Path End Structure</td>
</tr>
<tr>
<td>Table 10-3</td>
<td>PCI Device Path</td>
</tr>
<tr>
<td>Table 10-4</td>
<td>PCCARD Device Path</td>
</tr>
<tr>
<td>Table 10-5</td>
<td>MemoryMapped Device Path</td>
</tr>
<tr>
<td>Table 10-6</td>
<td>Vendor-Defined Device Path</td>
</tr>
<tr>
<td>Table 10-7</td>
<td>Controller Device Path</td>
</tr>
<tr>
<td>Table 10-8</td>
<td>BMC Device Path</td>
</tr>
<tr>
<td>Table 10-9</td>
<td>ACPI Device Path</td>
</tr>
<tr>
<td>Table 10-10</td>
<td>Expanded ACPI Device Path</td>
</tr>
<tr>
<td>Table 10-11</td>
<td>ACPI _ADR Device Path</td>
</tr>
<tr>
<td>Table 10-12</td>
<td>NVDIMM Device Path</td>
</tr>
<tr>
<td>Table 10-13</td>
<td>ATAPI Device Path</td>
</tr>
<tr>
<td>Table 10-14</td>
<td>SCSI Device Path</td>
</tr>
<tr>
<td>Table 10-15</td>
<td>Fibre Channel Device Path</td>
</tr>
<tr>
<td>Table 10-16</td>
<td>Fibre Channel Ex Device Path</td>
</tr>
<tr>
<td>Table 10-17</td>
<td>Fibre Channel Ex Device Path Example</td>
</tr>
<tr>
<td>Table 10-18</td>
<td>1394 Device Path</td>
</tr>
<tr>
<td>Table 10-19</td>
<td>USB Device Path</td>
</tr>
<tr>
<td>Table 10-20</td>
<td>USB Device Path Examples</td>
</tr>
<tr>
<td>Table 10-21</td>
<td>Another USB Device Path Example</td>
</tr>
<tr>
<td>Table 10-22</td>
<td>SATA Device Path</td>
</tr>
<tr>
<td>Table 10-23</td>
<td>USB WWID Device Path</td>
</tr>
<tr>
<td>Table 10-24</td>
<td>Device Logical Unit</td>
</tr>
<tr>
<td>Table 10-25</td>
<td>USB Class Device Path</td>
</tr>
<tr>
<td>Table 10-26</td>
<td>I2O Device Path</td>
</tr>
<tr>
<td>Table 10-27</td>
<td>MAC Address Device Path</td>
</tr>
<tr>
<td>Table 10-28</td>
<td>IPv4 Device Path</td>
</tr>
<tr>
<td>Table 10-29</td>
<td>IPv6 Device Path</td>
</tr>
<tr>
<td>Table 10-30</td>
<td>InfiniBand Device Path</td>
</tr>
<tr>
<td>Table 10-31</td>
<td>1394 Device Path</td>
</tr>
<tr>
<td>Table 10-32</td>
<td>Vendor-Defined Messaging Device Path</td>
</tr>
<tr>
<td>Table 10-33</td>
<td>UART Flow Control Messaging Device Path</td>
</tr>
<tr>
<td>Table 10-34</td>
<td>Messaging Device Path Structure</td>
</tr>
<tr>
<td>Table 10-35</td>
<td>Messaging Device Path Structure</td>
</tr>
<tr>
<td>Table 10-36</td>
<td>iSCSI Device Path</td>
</tr>
<tr>
<td>Table 10-37</td>
<td>IPv4 configuration</td>
</tr>
<tr>
<td>Table 10-38</td>
<td>IPv6 configuration</td>
</tr>
<tr>
<td>Table 10-39</td>
<td>NVM Express Namespace Device Path</td>
</tr>
<tr>
<td>Table 10-40</td>
<td>URI Device Path</td>
</tr>
<tr>
<td>Table 10-41</td>
<td>UFS Device Path</td>
</tr>
<tr>
<td>Table</td>
<td>Description</td>
</tr>
<tr>
<td>-------</td>
<td>-----------------------------------------------------------------------------</td>
</tr>
<tr>
<td>10-42</td>
<td>SD Device Path</td>
</tr>
<tr>
<td>10-43</td>
<td>Bluetooth Device Path</td>
</tr>
<tr>
<td>10-44</td>
<td>Wi-Fi Device Path</td>
</tr>
<tr>
<td>10-45</td>
<td>eMMC Device Path</td>
</tr>
<tr>
<td>10-46</td>
<td>EFI BluetoothLE Device Path</td>
</tr>
<tr>
<td>10-47</td>
<td>DNS Device Path</td>
</tr>
<tr>
<td>10-48</td>
<td>NVDIMM Namespace Device Path</td>
</tr>
<tr>
<td>10-49</td>
<td>Hard Drive Media Device Path</td>
</tr>
<tr>
<td>10-50</td>
<td>CD-ROM Media Device Path</td>
</tr>
<tr>
<td>10-51</td>
<td>Vendor-Defined Media Device Path</td>
</tr>
<tr>
<td>10-52</td>
<td>File Path Media Device Path</td>
</tr>
<tr>
<td>10-53</td>
<td>Media Protocol Media Device Path</td>
</tr>
<tr>
<td>10-54</td>
<td>PIWG Firmware Volume Device Path</td>
</tr>
<tr>
<td>10-55</td>
<td>PIWG Firmware Volume Device Path</td>
</tr>
<tr>
<td>10-56</td>
<td>Relative Offset Range</td>
</tr>
<tr>
<td>10-57</td>
<td>RAM Disk Device Path</td>
</tr>
<tr>
<td>10-58</td>
<td>BIOS Boot Specification Device Path</td>
</tr>
<tr>
<td>10-59</td>
<td>ACPI _CRS to EFI Device Path Mapping</td>
</tr>
<tr>
<td>10-60</td>
<td>ACPI _ADR to EFI Device Path Mapping</td>
</tr>
<tr>
<td>10-61</td>
<td>EFI Device Path Option Parameter Values</td>
</tr>
<tr>
<td>10-62</td>
<td>Device Node Table</td>
</tr>
<tr>
<td>12-1</td>
<td>Supported Unicode Control Characters</td>
</tr>
<tr>
<td>12-2</td>
<td>EFI Scan Codes for EFI_SIMPLE_TEXT_INPUT_PROTOCOL</td>
</tr>
<tr>
<td>12-3</td>
<td>EFI Scan Codes for EFI_SIMPLE_TEXT_INPUT_EX_PROTOCOL</td>
</tr>
<tr>
<td>12-4</td>
<td>EFI Cursor Location/Advance Rules</td>
</tr>
<tr>
<td>12-5</td>
<td>PS/2 Mouse Device Path</td>
</tr>
<tr>
<td>12-6</td>
<td>Serial Mouse Device Path</td>
</tr>
<tr>
<td>12-7</td>
<td>USB Mouse Device Path</td>
</tr>
<tr>
<td>12-8</td>
<td>Bit Operation Table</td>
</tr>
<tr>
<td>12-9</td>
<td>Attributes Definition Table</td>
</tr>
<tr>
<td>13-1</td>
<td>Tape Header Formats</td>
</tr>
<tr>
<td>13-2</td>
<td>PATA device formats to ports and port multiplier ports</td>
</tr>
<tr>
<td>13-3</td>
<td>Special programming considerations</td>
</tr>
<tr>
<td>14-1</td>
<td>PCI Configuration Address</td>
</tr>
<tr>
<td>14-2</td>
<td>QWORD Address Space Descriptor</td>
</tr>
<tr>
<td>14-3</td>
<td>End Tag</td>
</tr>
<tr>
<td>14-4</td>
<td>PCI Root Bridge Device Path for a Desktop System</td>
</tr>
<tr>
<td>14-5</td>
<td>PCI Root Bridge Device Path for Bridge #0 in a Server System</td>
</tr>
<tr>
<td>14-6</td>
<td>PCI Root Bridge Device Path for Bridge #1 in a Server System</td>
</tr>
<tr>
<td>14-7</td>
<td>PCI Root Bridge Device Path for Bridge #2 in a Server System</td>
</tr>
<tr>
<td>14-8</td>
<td>PCI Root Bridge Device Path for Bridge #3 in a Server System</td>
</tr>
<tr>
<td>14-9</td>
<td>PCI Root Bridge Device Path Using Expanded ACPI Device Path</td>
</tr>
<tr>
<td>14-10</td>
<td>QWORD Address Space Descriptor</td>
</tr>
</tbody>
</table>
Table 18-1 Debugport Messaging Device Path ......................................................................................... 880
Table 17-4 Payload-associated Messages and Descriptions ..................................................................... 843
Table 17-3 USB Port Features ................................................................................................................... 803
Table 17-2 Hub Port Change Status Bitmap .............................................................................................. 801
Table 17-1 USB Hub Port Status Bitmap ................................................................................................... 800
Table 15-7 Channel #3 of a PCI SCSI Controller behind a PCI Bridge ........................................................ 757
Table 15-6 Single Channel PCI SCSI Controller behind a PCI Bridge.......................................................... 756
Table 15-5 Single Channel PCI SCSI Controller ......................................................................................... 755
Table 15-4 InfiniBand Device Path Examples ............................................................................................ 754
Table 15-3 Fibre Channel Device Path Examples ...................................................................................... 753
Table 15-2 ATAPI Device Path Examples ................................................................................................... 752
Table 15-1 SCSI Device Path Examples...................................................................................................... 751
Table 14-12 PCI Device 7, Function 0 on PCI Root Bridge 0 ...................................................................... 725
Table 14-11 End Tag .................................................................................................................................. 722
Table 14-13 PCI Device 7, Function 0 behind PCI to PCI bridge ................................................................. 726
Table 14-14 Standard PCI Expansion ROM Header (Example from PCI Firmware Specification 3.0) ...... 728
Table 14-15 PCI Expansion ROM Code Types (Example from PCI Firmware Specification 3.0) .............. 728
Table 14-16 EFI PCI Expansion ROM Header ............................................................................................ 728
Table 14-17 Device Path for an EFI Driver loaded from PCI Option ROM ................................................. 730
Table 14-18 Recommended PCI Device Driver Layout .............................................................................. 736
Table 15-1 SCSI Device Path Examples .................................................................................................... 757
Table 15-2 ATAPI Device Path Examples ................................................................................................... 752
Table 15-3 Fibre Channel Device Path Examples ...................................................................................... 753
Table 15-4 InfiniBand Device Path Examples ............................................................................................ 754
Table 15-5 Single Channel PCI SCSI Controller ......................................................................................... 755
Table 15-6 Single Channel PCI SCSI Controller behind a PCI Bridge ........................................................ 756
Table 15-7 Channel #3 of a PCI SCSI Controller behind a PCI Bridge ........................................................ 757
Table 17-1 USB Hub Port Status Bitmap ................................................................................................... 800
Table 17-2 Hub Port Change Status Bitmap .............................................................................................. 801
Table 17-3 USB Port Features ................................................................................................................... 803
Table 17-4 Payload-associated Messages and Descriptions ..................................................................... 843
Table 18-1 Debugport Messaging Device Path .......................................................................................... 880
Table 19-1 Block Header Fields ................................................................................................................. 888
Table 21-1 General Purpose VM Registers ................................................................................................. 917
Table 21-2 Dedicated VM Registers .......................................................................................................... 917
Table 21-3 VM Flags Register .................................................................................................................... 918
Table 22-1 General Purpose VM Registers ................................................................................................. 917
Table 22-2 Dedicated VM Registers .......................................................................................................... 917
Table 22-3 VM Flags Register .................................................................................................................... 918
Table 22-4 Index Encoding ........................................................................................................................ 918
Table 22-5 Index Size in Index Encoding .................................................................................................... 919
Table 22-6 Opcode Byte Encoding ............................................................................................................ 922
Table 22-7 Operand Byte Encoding ........................................................................................................... 923
Table 22-8 ADD Instruction Encoding ....................................................................................................... 924
Table 22-9 AND Instruction Encoding ...................................................................................................... 925
Table 22-10 ASHR Instruction Encoding .................................................................................................. 926
Table 22-11 VM Version Format ................................................................................................................. 927
Table 22-12 BREAK Instruction Encoding .................................................................................................. 927
Table 22-13 CALL Instruction Encoding ................................................................................................. 929
Table 22-14 CMP Instruction Encoding ..................................................................................................... 930
Table 22-15 CMPI Instruction Encoding .................................................................................................. 931
Table 22-16 DIV Instruction Encoding ...................................................................................................... 932
Table 22-17 DIVU Instruction Encoding .................................................................................................... 933
Table 22-18 EXTNDB Instruction Encoding .............................................................................................. 934
Table 22-19 EXTNDD Instruction Encoding .............................................................................................. 935
Table 22-20 EXTNDW Instruction Encoding ............................................................................................ 936
Table 22-21 JMP Instruction Encoding ..................................................................................................... 938
Table 22-22 JMP8 Instruction Encoding ..................................................................................................... 939
<table>
<thead>
<tr>
<th>Table Number</th>
<th>Instruction Encoding</th>
</tr>
</thead>
<tbody>
<tr>
<td>22-23</td>
<td>LOADSP Instruction Encoding</td>
</tr>
<tr>
<td>22-24</td>
<td>MOD Instruction Encoding</td>
</tr>
<tr>
<td>22-25</td>
<td>MODU Instruction Encoding</td>
</tr>
<tr>
<td>22-26</td>
<td>MOV Instruction Encoding</td>
</tr>
<tr>
<td>22-27</td>
<td>MOVI Instruction Encoding</td>
</tr>
<tr>
<td>22-28</td>
<td>MOVIn Instruction Encoding</td>
</tr>
<tr>
<td>22-29</td>
<td>MOVn Instruction Encoding</td>
</tr>
<tr>
<td>22-30</td>
<td>MOVREL Instruction Encoding</td>
</tr>
<tr>
<td>22-31</td>
<td>MOVsn Instruction Encoding</td>
</tr>
<tr>
<td>22-32</td>
<td>MUL Instruction Encoding</td>
</tr>
<tr>
<td>22-33</td>
<td>MULU Instruction Encoding</td>
</tr>
<tr>
<td>22-34</td>
<td>NEG Instruction Encoding</td>
</tr>
<tr>
<td>22-35</td>
<td>NOT Instruction Encoding</td>
</tr>
<tr>
<td>22-36</td>
<td>OR Instruction Encoding</td>
</tr>
<tr>
<td>22-37</td>
<td>POP Instruction Encoding</td>
</tr>
<tr>
<td>22-38</td>
<td>POPn Instruction Encoding</td>
</tr>
<tr>
<td>22-39</td>
<td>PUSH Instruction Encoding</td>
</tr>
<tr>
<td>22-40</td>
<td>PUSHn Instruction Encoding</td>
</tr>
<tr>
<td>22-41</td>
<td>RET Instruction Encoding</td>
</tr>
<tr>
<td>22-42</td>
<td>SHL Instruction Encoding</td>
</tr>
<tr>
<td>22-43</td>
<td>SHR Instruction Encoding</td>
</tr>
<tr>
<td>22-44</td>
<td>STORESP Instruction Encoding</td>
</tr>
<tr>
<td>22-45</td>
<td>SUB Instruction Encoding</td>
</tr>
<tr>
<td>22-46</td>
<td>XOR Instruction Encoding</td>
</tr>
<tr>
<td>23-1</td>
<td>PUSH_GUID Instruction Encoding</td>
</tr>
<tr>
<td>23-2</td>
<td>PUSH_VERSION Instruction Encoding</td>
</tr>
<tr>
<td>23-3</td>
<td>DECLARE_VERSION _NAME Instruction Encoding</td>
</tr>
<tr>
<td>23-4</td>
<td>AND Instruction Encoding</td>
</tr>
<tr>
<td>23-5</td>
<td>OR Instruction Encoding</td>
</tr>
<tr>
<td>23-6</td>
<td>NOT Instruction Encoding</td>
</tr>
<tr>
<td>23-7</td>
<td>TRUE Instruction Encoding</td>
</tr>
<tr>
<td>23-8</td>
<td>FALSE Instruction Encoding</td>
</tr>
<tr>
<td>23-9</td>
<td>EQ Instruction Encoding</td>
</tr>
<tr>
<td>23-10</td>
<td>GT Instruction Encoding</td>
</tr>
<tr>
<td>23-11</td>
<td>GTE Instruction Encoding</td>
</tr>
<tr>
<td>23-12</td>
<td>LT Instruction Encoding</td>
</tr>
<tr>
<td>23-13</td>
<td>LTE Instruction Encoding</td>
</tr>
<tr>
<td>23-14</td>
<td>END Instruction Encoding</td>
</tr>
<tr>
<td>23-15</td>
<td>DECLARE_LENGTH Instruction Encoding</td>
</tr>
<tr>
<td>23-16</td>
<td>ESR and FMP Fields</td>
</tr>
<tr>
<td>24-1</td>
<td>PXE Tag Definitions for EFI</td>
</tr>
<tr>
<td>24-2</td>
<td>Destination IP Filter Operation</td>
</tr>
<tr>
<td>24-3</td>
<td>Destination UDP Port Filter Operation</td>
</tr>
<tr>
<td>Table</td>
<td>Description</td>
</tr>
<tr>
<td>-------</td>
<td>-------------</td>
</tr>
<tr>
<td>24-4</td>
<td>Source IP Filter Operation</td>
</tr>
<tr>
<td>24-5</td>
<td>Source UDP Port Filter Operation</td>
</tr>
<tr>
<td>29-1</td>
<td>DHCP4 Enumerations</td>
</tr>
<tr>
<td>29-2</td>
<td>Field Descriptions</td>
</tr>
<tr>
<td>29-3</td>
<td>Callback Return Values</td>
</tr>
<tr>
<td>30-1</td>
<td>Descriptions of Parameters in MTFTPv4 Packet Structures</td>
</tr>
<tr>
<td>30-2</td>
<td>Descriptions of Parameters in MTFTPv6 Packet Structures</td>
</tr>
<tr>
<td>30-3</td>
<td>MTFTP Packet OpCode Descriptions</td>
</tr>
<tr>
<td>30-4</td>
<td>MTFTP ERROR Packet_ErrorCode Descriptions</td>
</tr>
<tr>
<td>32-1</td>
<td>Generic Authentication Node Structure</td>
</tr>
<tr>
<td>32-2</td>
<td>CHAP Authentication Node Structure using RADIUS</td>
</tr>
<tr>
<td>32-3</td>
<td>CHAP Authentication Node Structure using Local Database</td>
</tr>
<tr>
<td>32-4</td>
<td>PE/COFF Certificates Types and UEFI Signature Database Certificate Types</td>
</tr>
<tr>
<td>32-5</td>
<td>Authorization process flow</td>
</tr>
<tr>
<td>33-1</td>
<td>Localization Issues</td>
</tr>
<tr>
<td>33-2</td>
<td>Information for Types of Storage</td>
</tr>
<tr>
<td>33-3</td>
<td>Common Control Codes for Font Display Information</td>
</tr>
<tr>
<td>33-4</td>
<td>Guidelines for UEFI System Fonts</td>
</tr>
<tr>
<td>33-5</td>
<td>Truth table: Mapping a single question to three configuration settings</td>
</tr>
<tr>
<td>33-6</td>
<td>Multiple configuration settings Example #2</td>
</tr>
<tr>
<td>33-7</td>
<td>Values</td>
</tr>
<tr>
<td>33-8</td>
<td>Package Types</td>
</tr>
<tr>
<td>33-9</td>
<td>Block Types</td>
</tr>
<tr>
<td>33-10</td>
<td>IFR Opcodes</td>
</tr>
<tr>
<td>33-11</td>
<td>VarStoreType Descriptions</td>
</tr>
<tr>
<td>33-12</td>
<td>Animation Block Types</td>
</tr>
<tr>
<td>35-1</td>
<td>Callback Behavior</td>
</tr>
<tr>
<td>36-1</td>
<td>Record values and descriptions</td>
</tr>
<tr>
<td>36-2</td>
<td>Standard values for access to configure the platform</td>
</tr>
<tr>
<td>37-1</td>
<td>EFI Hash Algorithms</td>
</tr>
<tr>
<td>37-2</td>
<td>Identical hash results</td>
</tr>
<tr>
<td>37-3</td>
<td>Algorithms that may be used with EFI_HASH2_PROTOCOL</td>
</tr>
<tr>
<td>37-4</td>
<td>Encryption algorithm properties</td>
</tr>
<tr>
<td>37-5</td>
<td>Details of Supported Signature Format</td>
</tr>
<tr>
<td>A-1</td>
<td>EFI GUID Format</td>
</tr>
<tr>
<td>A-2</td>
<td>Text representation relationships</td>
</tr>
<tr>
<td>B-1</td>
<td>EFI Scan Codes for EFI_SIMPLE_TEXT_INPUT_PROTOCOL</td>
</tr>
<tr>
<td>B-2</td>
<td>EFI Scan Codes for EFI_SIMPLE_TEXT_INPUT_EX_PROTOCOL</td>
</tr>
<tr>
<td>B-3</td>
<td>Control Sequences to Implement EFI_SIMPLE_TEXT_INPUT_PROTOCOL</td>
</tr>
<tr>
<td>C-1</td>
<td>Legacy Floppy Device Path</td>
</tr>
<tr>
<td>C-2</td>
<td>IDE Disk Device Path</td>
</tr>
<tr>
<td>C-3</td>
<td>Secondary Root PCI Bus with PCI to PCI Bridge Device Path</td>
</tr>
<tr>
<td>D-1</td>
<td>EFI_STATUS Code Ranges</td>
</tr>
</tbody>
</table>
Table N-31 Memory Error Record ...........................................................................................................2377
Table N-30 ARM Misc. Context System Register (Type 8) – Single Register Entry..................................2376
Table N-29 ARM AArch64 EL3 Context System Registers (Type 7) .........................................................2375
Table N-28 ARM AArch64 EL2 Context System Registers (Type 6) .........................................................2375
Table N-27 ARM AArch64 EL1 Context System Registers (Type 5) .........................................................2373
Table N-26 ARMv8 AArch64 GPRs (Type 4).............................................................................................2372
Table N-25 ARM AArch32 secure Context System Registers (Type 3) ....................................................2372
Table N-24 ARM AArch32 EL2 Context System Registers (Type 2) .........................................................2371
Table N-23 ARM AArch32 EL1 Context System Registers (Type 1) .........................................................2371
Table N-22 ARMv8 AArch32 GPRs (Type 0).............................................................................................2370
Table N-21 ARM Processor Error Context Information Header Structure ..............................................2370
Table N-20 ARM Bus Error Structure.......................................................................................................2368
Table N-19  ARM TLB Error Structure......................................................................................................2367
Table N-18 ARM Cache Error Structure...................................................................................................2366
Table N-17 ARM Processor Error Information Structure ......................................................................2365
Table N-16 ARM Processor Error Section.................................................................................................2363
Table N-15 X64 Register State..................................................................................................................2362
Table N-14 IA32 Register State .................................................................................................................2361
Table N-13 IA32/X64 Processor Context Information.............................................................................2360
Table N-12 IA32/X64 MS Check Field Description...................................................................................2360
Table N-11 IA32/X64 Bus Check Structure................................................................................................2358
Table N-10 IA32/X64 TLB Check Structure .............................................................................................2357
Table N-9 IA32/X64 Cache Check Structure ............................................................................................2356
Table N-8 IA32/X64 Processor Error Information Structure ...................................................................2355
Table N-7 Processor Error Record ...........................................................................................................2353
Table N-6 Processor Generic Error Section .............................................................................................2353
Table N-5 Section Descriptor ..................................................................................................................2349
Table N-4 Error Types..............................................................................................................................2348
Table N-3 Error Status Fields...................................................................................................................2348
Table N-2 Error Record Header ................................................................................................................2343
Table N-1 Error record header ................................................................................................................2343
Table M-1 Alias codes supported in addition to RFC 4646 .................................................................2341
Table L-1 Protocol Name changes...........................................................................................................2338
Table J-1 EBC Virtual Machine Opcode Summary ...................................................................................2334
Table E-5 UNDI CDB Field Definitions....................................................................................................2334
Table E-4 IPXE Structure Field Definitions ............................................................................................2334
Table E-3 Driver Types: Pros and Cons .................................................................................................2201
Table E-2 Referenced Specifications ........................................................................................................2198
Table E-1 Definitions ...............................................................................................................................2197
Table D-4 EFI_STATUS Warning Codes (High Bit Clear)...........................................................................2196
Table D-3 EFI_STATUS Error Codes (High Bit Set) ..................................................................................2194
Table D-2 EFI_STATUS Success Codes (High Bit Clear) ..........................................................................2194
Table D-1 EFI_STATUS Error record header ...........................................................................................2191
Table C-1 Pxe-Image Meta Fields ............................................................................................................2099
Table B-1 Sidetags .....................................................................................................................................2097
Table A-1 Driver Types: Pros and Cons .................................................................................................2002
Table A-2 Error Types ..............................................................................................................................2001
Table A-1 Definitions ...............................................................................................................................2000
Table M-2 Alias codes supported in addition to RFC 4646 .................................................................2000
Table K-1 VFIO Bus Mapping Options ....................................................................................................1995
Table J-1 EBC Virtual Machine Opcode Summary ...................................................................................1994
Table I-2 PCI Spanning Options ..............................................................................................................1986
Table H-1 PCI Bridge Mode .......................................................................................................................1986
Table G-1 PCI Memory Type ....................................................................................................................1984
Table F-1 PCI Access Fujian .....................................................................................................................1982
Table E-5 UNDI CDB Field Definitions....................................................................................................1981
Table D-5 EFI_STATUS Warning Codes (High Bit Clear)...........................................................................1980
Table D-4 EFI_STATUS Error Codes (High Bit Set) ..................................................................................1979
Table D-3 EFI_STATUS Error Codes (High Bit Set) ..................................................................................1977
Table D-2 EFI_STATUS Error Codes (High Bit Clear)...............................................................................1975
Table D-1 EFI_STATUS Error Codes (High Bit Clear)...............................................................................1974
Table C-2 Pxe-Image Meta Fields ............................................................................................................1973
Table B-2 Sidetags .....................................................................................................................................1972
Table A-3 Error Types ..............................................................................................................................1971
Table A-2 Error Types ..............................................................................................................................1970
Table A-1 Definitions ...............................................................................................................................1969
Table M-1 Alias codes supported in addition to RFC 4646 .................................................................1968
Table K-1 VFIO Bus Mapping Options ....................................................................................................1967
Table J-1 EBC Virtual Machine Opcode Summary ...................................................................................1966
Table I-2 PCI Spanning Options ..............................................................................................................1965
Table H-1 PCI Bridge Mode .......................................................................................................................1964
Table G-1 PCI Memory Type ....................................................................................................................1962
Table F-1 PCI Access Fujian .....................................................................................................................1961
Table E-5 UNDI CDB Field Definitions....................................................................................................1960
Table D-5 EFI_STATUS Warning Codes (High Bit Clear)...........................................................................1959
Table D-4 EFI_STATUS Error Codes (High Bit Set) ..................................................................................1958
Table D-3 EFI_STATUS Error Codes (High Bit Set) ..................................................................................1957
Table D-2 EFI_STATUS Error Codes (High Bit Clear)...............................................................................1955
Table D-1 EFI_STATUS Error Codes (High Bit Clear)...............................................................................1954
Table C-2 Pxe-Image Meta Fields ............................................................................................................1953
Table B-2 Sidetags .....................................................................................................................................1952
Table A-3 Error Types ..............................................................................................................................1951
Table A-2 Error Types ..............................................................................................................................1950
Table A-1 Definitions ...............................................................................................................................1949
Table N-32 Memory Error Record 2 ................................................................. 2379
Table N-33 PCI Express Error Record .......................................................... 2381
Table N-34 PCI/PCI-X Bus Error Section ..................................................... 2382
Table N-35 PCI/PCI-X Component Error Section ...................................... 2383
Table N-36 Firmware Error Record Reference ......................................... 2384
Table N-37 DMAr Generic Errors ................................................................. 2385
Table N-38 Intel® VT for Directed I/O specific DMAr Errors .................... 2386
Table N-39 IOMMU-specific DMAr Errors .................................................. 2387
Table N-40 CCIX PER Log Error Record .................................................... 2388
Table N-41 Table 94. CXL Protocol Error Section ...................................... 2388
Table N-42 CXL Component Event Log Record ........................................ 2390
Table O-1 UEFI Table Structure ................................................................. 2391
Table O-2 SMM Communication ACPI Table ............................................ 2391
# List of Figures

<table>
<thead>
<tr>
<th>Figure</th>
<th>Title</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>1-1</td>
<td>UEFI Conceptual Overview</td>
<td>7</td>
</tr>
<tr>
<td>2-1</td>
<td>Booting Sequence</td>
<td>14</td>
</tr>
<tr>
<td>2-2</td>
<td>Stack after <code>AddressOfEntryPoint</code> Called, IA- 32</td>
<td>24</td>
</tr>
<tr>
<td>2-3</td>
<td>Stack after <code>AddressOfEntryPoint</code> Called, Itanium-based Systems</td>
<td>26</td>
</tr>
<tr>
<td>2-4</td>
<td>Construction of a Protocol</td>
<td>46</td>
</tr>
<tr>
<td>2-5</td>
<td>Desktop System</td>
<td>51</td>
</tr>
<tr>
<td>2-6</td>
<td>Server System</td>
<td>51</td>
</tr>
<tr>
<td>2-7</td>
<td>Image Handle</td>
<td>54</td>
</tr>
<tr>
<td>2-8</td>
<td>Driver Image Handle</td>
<td>55</td>
</tr>
<tr>
<td>2-9</td>
<td>Host Bus Controllers</td>
<td>56</td>
</tr>
<tr>
<td>2-10</td>
<td>PCI Root Bridge Device Handle</td>
<td>56</td>
</tr>
<tr>
<td>2-11</td>
<td>Connecting Device Drivers</td>
<td>57</td>
</tr>
<tr>
<td>2-12</td>
<td>Connecting Bus Drivers</td>
<td>58</td>
</tr>
<tr>
<td>2-13</td>
<td>Child Device Handle with a Bus Specific Override</td>
<td>59</td>
</tr>
<tr>
<td>2-14</td>
<td>Software Service Relationships</td>
<td>61</td>
</tr>
<tr>
<td>5-1</td>
<td>MBR Disk Layout with legacy MBR example</td>
<td>117</td>
</tr>
<tr>
<td>5-2</td>
<td>GPT disk layout with protective MBR</td>
<td>119</td>
</tr>
<tr>
<td>5-3</td>
<td>GPT disk layout with protective MBR on a disk with capacity &gt; LBA 0xffffffff</td>
<td>119</td>
</tr>
<tr>
<td>5-4</td>
<td>GUID Partition Table (GPT) example</td>
<td>121</td>
</tr>
<tr>
<td>6-1</td>
<td>The BTT Layout in a BTT Arena</td>
<td>128</td>
</tr>
<tr>
<td>6-2</td>
<td>A BTT With Multiple Arenas in a Large Namespace</td>
<td>129</td>
</tr>
<tr>
<td>6-3</td>
<td>Cyclic Sequence Numbers for Flog Entries</td>
<td>134</td>
</tr>
<tr>
<td>6-4</td>
<td>BTT Read Path Overview</td>
<td>141</td>
</tr>
<tr>
<td>6-5</td>
<td>BTT Write Path Overview</td>
<td>142</td>
</tr>
<tr>
<td>7-1</td>
<td>Device Handle to Protocol Handler Mapping</td>
<td>177</td>
</tr>
<tr>
<td>7-2</td>
<td>Handle Database</td>
<td>178</td>
</tr>
<tr>
<td>8-1</td>
<td>Scatter-Gather List of EFI_CAPSULE_BLOCK_DESCRIPTOR Structures</td>
<td>277</td>
</tr>
<tr>
<td>10-1</td>
<td>Text to Binary Conversion</td>
<td>341</td>
</tr>
<tr>
<td>10-2</td>
<td>Binary to Text Conversion</td>
<td>341</td>
</tr>
<tr>
<td>10-3</td>
<td>Device Path Text Representation</td>
<td>343</td>
</tr>
<tr>
<td>10-4</td>
<td>Text Device Node Names</td>
<td>343</td>
</tr>
<tr>
<td>10-5</td>
<td>Device Node Option Names</td>
<td>344</td>
</tr>
<tr>
<td>11-1</td>
<td>Driver Health Status States</td>
<td>416</td>
</tr>
<tr>
<td>12-1</td>
<td>Serial Device Identification Driver Relationships</td>
<td>477</td>
</tr>
<tr>
<td>12-2</td>
<td>Software BLT Buffer</td>
<td>484</td>
</tr>
<tr>
<td>13-1</td>
<td>Nesting of Legacy MBR Partition Records</td>
<td>507</td>
</tr>
<tr>
<td>13-2</td>
<td>Cyclic Sequence Numbers in Label Index Block</td>
<td>632</td>
</tr>
<tr>
<td>14-1</td>
<td>Host Bus Controllers</td>
<td>650</td>
</tr>
</tbody>
</table>
Figure 14-2 Device Handle for a PCI Root Bridge Controller ................................................................. 651
Figure 14-3 Desktop System with One PCI Root Bridge ................................................................. 651
Figure 14-4 Server System with Four PCI Root Bridges ................................................................. 652
Figure 14-5 Server System with Two PCI Segments ..................................................................... 653
Figure 14-6 Server System with Two PCI Host Buses ..................................................................... 653
Figure 14-7 Image Handle .................................................................................................................. 683
Figure 14-8 PCI Driver Image Handle .............................................................................................. 684
Figure 14-9 PCI Host Bus Controller .............................................................................................. 685
Figure 14-10 Device Handle for a PCI Host Bus Controller ............................................................. 686
Figure 14-11 Physical PCI Bus Structure ...................................................................................... 687
Figure 14-12 Connecting a PCI Bus Driver ...................................................................................... 688
Figure 14-13 Child Handle Created by a PCI Bus Driver ................................................................. 688
Figure 14-14 Connecting a PCI Device Driver .................................................................................. 691
Figure 14-15 Unsigned PCI Driver Image Layout ............................................................................ 731
Figure 14-16 Signed and Compressed PCI Driver Image Flow ....................................................... 732
Figure 14-17 Signed and Compressed PCI Driver Image Layout ..................................................... 733
Figure 14-18 Signed but not Compressed PCI Driver Image Flow ................................................... 734
Figure 14-19 Signed and Uncompressed PCI Driver Image Layout ................................................... 735
Figure 15-1 Device Handle for a SCSI Bus Controller ..................................................................... 739
Figure 15-2 Child Handle Created by a SCSI Bus Driver ................................................................. 739
Figure 17-1 Software Triggered State Transitions of a USB Host Controller ..................................... 785
Figure 17-2 USB Bus Controller Handle ......................................................................................... 806
Figure 17-3 Sequence of Operations with Endpoint Policy Changes .............................................. 855
Figure 18-1 Debug Support Table Indirection and Pointer Usage ..................................................... 882
Figure 19-1 Bit Sequence of Compressed Data .................................................................................. 886
Figure 19-2 Compressed Data Structure ......................................................................................... 887
Figure 19-3 Block Structure .............................................................................................................. 888
Figure 19-4 Block Body .................................................................................................................... 890
Figure 19-5 String Info Log Search Tree ............................................................................................ 892
Figure 19-6 Node Split ....................................................................................................................... 894
Figure 23-1 Firmware Image with no Authentication Support ............................................................ 983
Figure 23-2 Firmware Image with Authentication Support .............................................................. 984
Figure 23-3 Firmware Image with Dependency/Authentication Support .......................................... 984
Figure 23-4 Optional Scatter-Gather Construction of Capsule Submitted to UpdateCapsule() .......... 1003
Figure 23-5 Capsule Header and Firmware Management Capsule Header ...................................... 1004
Figure 23-6 Firmware Management and Firmware Image Management headers ............................ 1005
Figure 24-1 IPv6-based PXE boot .................................................................................................... 1076
Figure 24-2 netboot6 (DHCPS6 and ProxyDHCPS6 reside on the same server) ................................. 1078
Figure 24-3 IPv6-based PXE boot (DHCPS6 and ProxyDHCPS6 reside on the different server) .......... 1079
Figure 24-4 HTTP Boot Network Topology Concept – Corporate Environment ......................... 1118
Figure 24-5 HTTP Boot Network Topology Concept 2 – Home environments ............................. 1119
Figure 33-29 Typical Forms Processor Decisions Necessitating a Callback (1) ........................................1780
Figure 33-30 Typical Forms Processor Decisions Necessitating a Callback (2) ........................................1781
Figure 33-31 Typical Forms Processor Decisions Necessitating a Callback (3) ........................................1782
Figure 33-32 Driver Model Interactions ..................................................................................................1783
Figure 33-33 Managing Human Interface Components ..........................................................................1784
Figure 33-34 EFI IFR Form set configuration ..........................................................................................1785
Figure 33-35 EFI IFR Form Set question changes ....................................................................................1786
Figure 33-36 Glyph Information Encoded in Blocks ..............................................................................1789
Figure 33-37 Glyph Block Processing ......................................................................................................1798
Figure 33-38 EFI_HII_GIB_T_GLYPH_VARIABLITY Glyph Drawing Processing ........................................1806
Figure 33-39 String Information Encoded in Blocks ..............................................................................1809
Figure 33-40 String Block Processing: Base Processing .........................................................................1811
Figure 33-41 String Block Processing: SCSU Processing ......................................................................1812
Figure 33-42 String Block Processing: UTF Processing .........................................................................1813
Figure 33-43 Image Information Encoded in Blocks .............................................................................1824
Figure 33-44 Palette Structure of a Black & White, One-Bit Image .........................................................1838
Figure 33-45 Palette Structure of a Four-Bit Image ................................................................................1839
Figure 33-46 Palette Structure of a Four-Bit, Six-Color Image ...............................................................1840
Figure 33-47 Simple Binary Object .......................................................................................................1841
Figure 33-48 Password Flowchart (part one) .........................................................................................1885
Figure 33-49 Password Flowchart (part two) .........................................................................................1886
Figure 33-50 Animation Information Encoded in Blocks ....................................................................1918
Figure 34-1 Glyph Example ..................................................................................................................1943
Figure 34-2 How EFI_HII_IMAGE_EX_PROTOCOL uses EFI_HII_IMAGE_DECODER_PROTOCOL ..........1967
Figure 34-3 Keyboard Layout ...............................................................................................................1989
Figure 36-1 User Identity .......................................................................................................................2035
Figure 37-1 Hash workflow ..................................................................................................................2089
Figure C-1 Example Computer System ...............................................................................................2188
Figure C-2 Partial ACPI Name Space for Example System ..................................................................2189
Figure C-3 EFI Device Path Displayed As a Name Space ..................................................................2193
Figure E-1 Network Stacks with Three Classes of Drivers ................................................................2201
Figure E-2 IPXE Structures for H/W and S/W UNDI ..........................................................................2202
Figure E-3 Issuing UNDI Commands ..................................................................................................2206
Figure E-4 UNDI Command Descriptor Block (CDB) .........................................................................2207
Figure E-5 Storage Types .....................................................................................................................2211
Figure E-6 UNDI States, Transitions & Valid Commands ....................................................................2232
Figure E-7 Linked CDBs .......................................................................................................................2233
Figure E-8 Queued CDBs .....................................................................................................................2234
Figure N-1 Error Record Format .........................................................................................................2342
1 - Introduction

This Unified Extensible Firmware Interface (UEFI) Specification describes an interface between the operating system (OS) and the platform firmware. UEFI was preceded by the Extensible Firmware Interface Specification 1.10 (EFI). As a result, some code and certain protocol names retain the EFI designation. Unless otherwise noted, EFI designations in this specification may be assumed to be part of UEFI.

The interface is in the form of data tables that contain platform-related information, and boot and runtime service calls that are available to the OS loader and the OS. Together, these provide a standard environment for booting an OS. This specification is designed as a pure interface specification. As such, the specification defines the set of interfaces and structures that platform firmware must implement. Similarly, the specification defines the set of interfaces and structures that the OS may use in booting. How either the firmware developer chooses to implement the required elements or the OS developer chooses to make use of those interfaces and structures is an implementation decision left for the developer.

The intent of this specification is to define a way for the OS and platform firmware to communicate only information necessary to support the OS boot process. This is accomplished through a formal and complete abstract specification of the software-visible interface presented to the OS by the platform and firmware.

Using this formal definition, a shrink-wrap OS intended to run on platforms compatible with supported processor specifications will be able to boot on a variety of system designs without further platform or OS customization. The definition will also allow for platform innovation to introduce new features and functionality that enhance platform capability without requiring new code to be written in the OS boot sequence.

Furthermore, an abstract specification opens a route to replace legacy devices and firmware code over time. New device types and associated code can provide equivalent functionality through the same defined abstract interface, again without impact on the OS boot support code.

The specification is applicable to a full range of hardware platforms from mobile systems to servers. The specification provides a core set of services along with a selection of protocol interfaces. The selection of protocol interfaces can evolve over time to be optimized for various platform market segments. At the same time, the specification allows maximum extensibility and customization abilities for OEMs to allow differentiation. In this, the purpose of UEFI is to define an evolutionary path from the traditional “PC-AT”-style boot world into a legacy-API free environment.

1.1 UEFI Driver Model Extensions

Access to boot devices is provided through a set of protocol interfaces. One purpose of the UEFI Driver Model is to provide a replacement for “PC-AT”-style option ROMs. It is important to point out that drivers written to the UEFI Driver Model are designed to access boot devices in the preboot environment. They are not designed to replace the high-performance, OS-specific drivers.

The UEFI Driver Model is designed to support the execution of modular pieces of code, also known as drivers, that run in the preboot environment. These drivers may manage or control hardware buses and devices on the platform, or they may provide some software-derived, platform-specific service.
The UEFI Driver Model also contains information required by UEFI driver writers to design and implement any combination of bus drivers and device drivers that a platform might need to boot a UEFI-compliant OS.

The UEFI Driver Model is designed to be generic and can be adapted to any type of bus or device. The UEFI Specification describes how to write PCI bus drivers, PCI device drivers, USB bus drivers, USB device drivers, and SCSI drivers. Additional details are provided that allow UEFI drivers to be stored in PCI option ROMs, while maintaining compatibility with legacy option ROM images.

One of the design goals in the UEFI Specification is keeping the driver images as small as possible. However, if a driver is required to support multiple processor architectures, a driver object file would also be required to be shipped for each supported processor architecture. To address this space issue, this specification also defines the EFI Byte Code Virtual Machine. A UEFI driver can be compiled into a single EFI Byte Code object file. UEFI Specification-compliant firmware must contain an EFI Byte Code interpreter. This allows a single EFI Byte Code object file that supports multiple processor architectures to be shipped. Another space saving technique is the use of compression. This specification defines compression and decompression algorithms that may be used to reduce the size of UEFI Drivers, and thus reduce the overhead when UEFI Drivers are stored in ROM devices.

The information contained in the UEFI Specification can be used by OSVs, IHVs, OEMs, and firmware vendors to design and implement firmware conforming to this specification, drivers that produce standard protocol interfaces, and operating system loaders that can be used to boot UEFI-compliant operating systems.

### 1.2 Organization

The high-level organization of this specification is as follows:

<table>
<thead>
<tr>
<th>Section(s)</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Introduction / Overview</td>
<td>Introduces the UEFI Specification, and describes the major components of UEFI.</td>
</tr>
<tr>
<td>Boot Manager</td>
<td>Manager used to load drivers and applications written to this specification.</td>
</tr>
<tr>
<td>EFI System Table and Partitions</td>
<td>Describes an EFI System Table that is passed to every compliant driver and application, and defines a GUID-based partitioning scheme.</td>
</tr>
<tr>
<td>Block Transition Table</td>
<td>A layout and set of rules for doing block I/O that provide powerfail write atomicity of a single block.</td>
</tr>
<tr>
<td>Boot Services</td>
<td>Contains the definitions of the fundamental services that are present in a UEFI-compliant system before an OS is booted.</td>
</tr>
<tr>
<td>Runtime Services</td>
<td>Contains definitions for the fundamental services that are present in a compliant system before and after an OS is booted.</td>
</tr>
<tr>
<td>Section(s)</td>
<td>Description</td>
</tr>
<tr>
<td>------------------------------------</td>
<td>-------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------</td>
</tr>
</tbody>
</table>
| Protocols                          | • The EFI Loaded Image Protocol describes a UEFI Image that has been loaded into memory.  
• The Device Path Protocol provides the information needed to construct and manage device paths in the UEFI environment.  
• The UEFI Driver Model describes a set of services and protocols that apply to every bus and device type.  
• The Console Support Protocol defines I/O protocols that handle input and output of text-based information intended for the system user while executing in the boot services environment.  
• The Media Access Protocol defines the Load File protocol, file system format and media formats for handling removable media.  
• PCI Bus Support Protocols define PCI Bus Drivers, PCI Device Drivers, and PCI Option ROM layouts. The protocols described include the PCI Root Bridge I/O Protocol and the PCI I/O Protocol.  
• SCSI Driver Models and Bus support defines the SCSI I/O Protocol and the Extended SCSI Pass Thru Protocol that is used to abstract access to a SCSI channel that is produced by a SCSI host controller.  
• The iSCSI protocol defines a transport for SCSI data over TCP/IP.  
• The USB Support Protocol defines USB Bus Drivers and USB Device Drivers.  
• Debugger Support Protocols describe an optional set of protocols that provide the services required to implement a source-level debugger for the UEFI environment.  
• The Compression Algorithm Specification describes the compression/decompression algorithm in detail, plus a standard EFI decompression interface for use at boot time.  
• ACPI Protocols may be used to install or remove an ACPI table from a platform.  
• String Services: the Unicode Collation protocol allows code running in the boot services environment to perform lexical comparison functions on Unicode strings for given languages; the Regular Expression Protocol is used to match Unicode strings against Regular Expression patterns.                                         |
| EFI Byte Code Virtual Machine      | Defines the EFI Byte Code virtual processor and its instruction set. It also defines how EBC object files are loaded into memory, and the mechanism for transitioning from native code to EBC code and back to native code. |
| Firmware Update and Reporting      | Provides an abstraction for devices to provide firmware management support.                                                                                                                                   |
| Network Protocols                  | • SNP, PXE, BIS, and HTTP Boot protocols define the protocols that provide access to network devices while executing in the UEFI boot services environment.  
• Managed Network protocols define the EFI Managed Network Protocol, which provides raw (unformatted) asynchronous network packet I/O services and Managed Network Service Binding Protocol, used to locate communication devices that are supported by an MNP driver.  
• VLAN, EAP, Wi-Fi and Supplicant protocols define a protocol that is to provide a manageability interface for VLAN configurations.  
• Bluetooth protocol definitions.  
• ARP, DHCP, DNS, HTTP, and REST protocols define the EFI Address Resolution Protocol (ARP) Protocol Interface and the EFI DHCPv4 Protocol.  
• UDP and MTFTP protocols define the EFI UD Pv4 (User Datagram Protocol version 4) Protocol that interfaces over the EFI IPv4 Protocol and defines the EFI MTFTPv4 Protocol interface that is built on the EFI UDPv4 Protocol. |
| Secure Boot and Driver Signing     | Describes Secure Boot and a means of generating a digital signature for UEFI.                                                                                                                                    |
| Human Interface Infrastructure (HII)| • Defines the core code and services that are required for an implementation of the Human Interface Infrastructure (HII), including basic mechanisms for managing user input and code definitions for related protocols.  
• Describes the data and APIs used to manage the system’s configuration: the actual data that describes the knobs and settings. |
1.3 Goals

The “PC-AT” boot environment presents significant challenges to innovation within the industry. Each new platform capability or hardware innovation requires firmware developers to craft increasingly complex solutions, and often requires OS developers to make changes to their boot code before customers can benefit from the innovation. This can be a time-consuming process requiring a significant investment of resources.

The primary goal of the UEFI specification is to define an alternative boot environment that can alleviate some of these considerations. In this goal, the specification is similar to other existing boot specifications. The main properties of this specification can be summarized by these attributes:

- **Coherent, scalable platform environment.** The specification defines a complete solution for the firmware to describe all platform features and surface platform capabilities to the OS.
during the boot process. The definitions are rich enough to cover a range of contemporary processor designs.

• **Abstraction of the OS from the firmware.** The specification defines interfaces to platform capabilities. Through the use of abstract interfaces, the specification allows the OS loader to be constructed with far less knowledge of the platform and firmware that underlie those interfaces. The interfaces represent a well-defined and stable boundary between the underlying platform and firmware implementation and the OS loader. Such a boundary allows the underlying firmware and the OS loader to change provided both limit their interactions to the defined interfaces.

• **Reasonable device abstraction free of legacy interfaces.** “PC-AT” BIOS interfaces require the OS loader to have specific knowledge of the workings of certain hardware devices. This specification provides OS loader developers with something different: abstract interfaces that make it possible to build code that works on a range of underlying hardware devices without having explicit knowledge of the specifics for each device in the range.

• **Abstraction of Option ROMs from the firmware.** This specification defines interfaces to platform capabilities including standard bus types such as PCI, USB, and SCSI. The list of supported bus types may grow over time, so a mechanism to extend to future bus types is included. These defined interfaces, and the ability to extend to future bus types, are components of the UEFI **Driver Model**. One purpose of the UEFI **Driver Model** is to solve a wide range of issues that are present in existing “PC-AT” option ROMs. Like OS loaders, drivers use the abstract interfaces so device drivers and bus drivers can be constructed with far less knowledge of the platform and firmware that underlie those interfaces.

• **Architecturally shareable system partition.** Initiatives to expand platform capabilities and add new devices often require software support. In many cases, when these platform innovations are activated before the OS takes control of the platform, they must be supported by code that is specific to the platform rather than to the customer’s choice of OS. The traditional approach to this problem has been to embed code in the platform during manufacturing (for example, in flash memory devices). Demand for such persistent storage is increasing at a rapid rate. This specification defines persistent store on large mass storage media types for use by platform support code extensions to supplement the traditional approach. The definition of how this works is made clear in the specification to ensure that firmware developers, OEMs, operating system vendors, and perhaps even third parties can share the space safely while adding to platform capability.

Defining a boot environment that delivers these attributes could be accomplished in many ways. Indeed, several alternatives, perhaps viable from an academic point of view, already existed at the time this specification was written. These alternatives, however, typically presented high barriers to entry given the current infrastructure capabilities surrounding supported processor platforms. This specification is intended to deliver the attributes listed above, while also recognizing the unique needs of an industry that has considerable investment in compatibility and a large installed base of systems that cannot be abandoned summarily. These needs drive the requirements for the additional attributes embodied in this specification:

• **Evolutionary, not revolutionary.** The interfaces and structures in the specification are designed to reduce the burden of an initial implementation as much as possible. While care has been taken to ensure that appropriate abstractions are maintained in the interfaces themselves, the design also ensures that reuse of BIOS code to implement the interfaces is possible with a
minimum of additional coding effort. In other words, on PC-AT platforms the specification can be implemented initially as a thin interface layer over an underlying implementation based on existing code. At the same time, introduction of the abstract interfaces provides for migration away from legacy code in the future. Once the abstraction is established as the means for the firmware and OS loader to interact during boot, developers are free to replace legacy code underneath the abstract interfaces at leisure. A similar migration for hardware legacy is also possible. Since the abstractions hide the specifics of devices, it is possible to remove underlying hardware, and replace it with new hardware that provides improved functionality, reduced cost, or both. Clearly this requires that new platform firmware be written to support the device and present it to the OS loader via the abstract interfaces. However, without the interface abstraction, removal of the legacy device might not be possible at all.

- **Compatibility by design.** The design of the system partition structures also preserves all the structures that are currently used in the “PC-AT” boot environment. Thus, it is a simple matter to construct a single system that is capable of booting a legacy OS or an EFI-aware OS from the same disk.

- **Simplifies addition of OS-neutral platform value-add.** The specification defines an open, extensible interface that lends itself to the creation of platform “drivers.” These may be analogous to OS drivers, providing support for new device types during the boot process, or they may be used to implement enhanced platform capabilities, such as fault tolerance or security. Furthermore, this ability to extend platform capability is designed into the specification from the outset. This is intended to help developers avoid many of the frustrations inherent in trying to squeeze new code into the traditional BIOS environment. As a result of the inclusion of interfaces to add new protocols, OEMs or firmware developers have an infrastructure to add capability to the platform in a modular way. Such drivers may potentially be implemented using high-level coding languages because of the calling conventions and environment defined in the specification. This in turn may help to reduce the difficulty and cost of innovation. The option of a system partition provides an alternative to nonvolatile memory storage for such extensions.

- **Built on existing investment.** Where possible, the specification avoids redefining interfaces and structures in areas where existing industry specifications provide adequate coverage. For example, the ACPI specification provides the OS with all the information necessary to discover and configure platform resources. Again, this philosophical choice for the design of the specification is intended to keep barriers to its adoption as low as possible.

### 1.4 Target Audience

This document is intended for the following readers:

- IHVs and OEMs who will be implementing UEFI drivers.
- OEMs who will be creating supported processor platforms intended to boot shrink-wrap operating systems.
- BIOS developers, either those who create general-purpose BIOS and other firmware products or those who modify these products for use in supported processor-based products.
- Operating system developers who will be adapting their shrink-wrap operating system products to run on supported processor-based platforms.
1.5 UEFI Design Overview

The design of UEFI is based on the following fundamental elements:

- **Reuse of existing table-based interfaces.** In order to preserve investment in existing infrastructure support code, both in the OS and firmware, a number of existing specifications that are commonly implemented on platforms compatible with supported processor specifications must be implemented on platforms wishing to comply with the UEFI specification. (For additional information, see Appendix Q: References.)

- **System partition.** The System partition defines a partition and file system that are designed to allow safe sharing between multiple vendors, and for different purposes. The ability to include a separate, sharable system partition presents an opportunity to increase platform value-add without significantly growing the need for nonvolatile platform memory.

- **Boot services.** Boot services provide interfaces for devices and system functionality that can be used during boot time. Device access is abstracted through “handles” and “protocols.” This facilitates reuse of investment in existing BIOS code by keeping underlying implementation requirements out of the specification without burdening the consumer accessing the device.

- **Runtime services.** A minimal set of runtime services is presented to ensure appropriate abstraction of base platform hardware resources that may be needed by the OS during its normal operations.

*Figure 1-1* shows the principal components of UEFI and their relationship to platform hardware and OS software.

---

![Figure 1-1 UEFI Conceptual Overview](image-url)

*Figure 1-1 UEFI Conceptual Overview*

*Figure 1-1* illustrates the interactions of the various components of an UEFI specification-compliant system that are used to accomplish platform and OS boot.
The platform firmware is able to retrieve the OS loader image from the System Partition. The specification provides for a variety of mass storage device types including disk, CD-ROM, and DVD as well as remote boot via a network. Through the extensible protocol interfaces, it is possible to add other boot media types, although these may require OS loader modifications if they require use of protocols other than those defined in this document.

Once started, the OS loader continues to boot the complete operating system. To do so, it may use the EFI boot services and interfaces defined by this or other required specifications to survey, comprehend, and initialize the various platform components and the OS software that manages them. EFI runtime services are also available to the OS loader during the boot phase.

1.6 UEFI Driver Model

This section describes the goals of a driver model for firmware conforming to this specification. The goal is for this driver model to provide a mechanism for implementing bus drivers and device drivers for all types of buses and devices. At the time of writing, supported bus types include PCI, USB, and so on.

As hardware architectures continue to evolve, the number and types of buses present in platforms are increasing. This trend is especially true in high-end servers. However, a more diverse set of bus types is being designed into desktop and mobile systems and even some embedded systems. This increasing complexity means that a simple method for describing and managing all the buses and devices in a platform is required in the preboot environment. The UEFI Driver Model provides this simple method in the form of protocols services and boot services.

1.6.1 UEFI Driver Model Goals

The UEFI Driver Model has the following goals:

- **Compatible** – Drivers conforming to this specification must maintain compatibility with the EFI 1.10 Specification and the UEFI Specification. This means that the UEFI Driver Model takes advantage of the extensibility mechanisms in the UEFI 2.0 Specification to add the required functionality.

- **Simple** – Drivers that conform to this specification must be simple to implement and simple to maintain. The UEFI Driver Model must allow a driver writer to concentrate on the specific device for which the driver is being developed. A driver should not be concerned with platform policy or platform management issues. These considerations should be left to the system firmware.

- **Scalable** – The UEFI Driver Model must be able to adapt to all types of platforms. These platforms include embedded systems, mobile, and desktop systems, as well as workstations and servers.

- **Flexible** – The UEFI Driver Model must support the ability to enumerate all the devices, or to enumerate only those devices required to boot the required OS. The minimum device enumeration provides support for more rapid boot capability, and the full device enumeration provides the ability to perform OS installations, system maintenance, or system diagnostics on any boot device present in the system.

- **Extensible** – The UEFI Driver Model must be able to extend to future bus types as they are defined.
• **Portable** – Drivers written to the UEFI Driver Model must be portable between platforms and between supported processor architectures.

• **Interoperable** – Drivers must coexist with other drivers and system firmware and must do so without generating resource conflicts.

• **Describe complex bus hierarchies** – The UEFI Driver Model must be able to describe a variety of bus topologies from very simple single bus platforms to very complex platforms containing many buses of various types.

• **Small driver footprint** – The size of executables produced by the UEFI Driver Model must be minimized to reduce the overall platform cost. While flexibility and extensibility are goals, the additional overhead required to support these must be kept to a minimum to prevent the size of firmware components from becoming unmanageable.

• **Address legacy option rom issues** – The UEFI Driver Model must directly address and solve the constraints and limitations of legacy option ROMs. Specifically, it must be possible to build add-in cards that support both UEFI drivers and legacy option ROMs, where such cards can execute in both legacy BIOS systems and UEFI-conforming platforms, without modifications to the code carried on the card. The solution must provide an evolutionary path to migrate from legacy option ROMs driver to UEFI drivers.

### 1.6.2 Legacy Option ROM Issues

This idea of supporting a driver model came from feedback on the UEFI Specification that provided a clear, market-driven requirement for an alternative to the legacy option ROM (sometimes also referred to as an expansion ROM). The perception is that the advent of the UEFI Specification represents a chance to escape the limitations implicit in the construction and operation of legacy option ROM images by replacing them with an alternative mechanism that works within the framework of the UEFI Specification.

### 1.7 Migration Requirements

Migration requirements cover the transition period from initial implementation of this specification to a future time when all platforms and operating systems implement to this specification. During this period, two major compatibility considerations are important:

- The ability to continue booting legacy operating systems;
- The ability to implement UEFI on existing platforms by reusing as much existing firmware code to keep development resource and time requirements to a minimum.

### 1.7.1 Legacy Operating System Support

The UEFI specification represents the preferred means for a shrink-wrap OS and firmware to communicate during the boot process. However, choosing to make a platform that complies with this specification in no way precludes a platform from also supporting existing legacy OS binaries that have no knowledge of the UEFI specification.

The UEFI specification does not restrict a platform designer who chooses to support both the UEFI specification and a more traditional “PC-AT” boot infrastructure. If such a legacy infrastructure is to be implemented, it should be developed in accordance with existing industry practice that is defined outside the scope of this specification. The choice of legacy operating systems that are supported on any given platform is left to the manufacturer of that platform.
1.7.2 Supporting the UEFI Specification on a Legacy Platform

The UEFI specification has been carefully designed to allow for existing systems to be extended to support it with a minimum of development effort. In particular, the abstract structures and services defined in the UEFI specification can all be supported on legacy platforms.

For example, to accomplish such support on an existing and supported 32-bit-based platform that uses traditional BIOS to support operating system boot, an additional layer of firmware code would need to be provided. This extra code would be required to translate existing interfaces for services and devices into support for the abstractions defined in this specification.

1.8 Conventions Used in this Document

This document uses typographic and illustrative conventions described below.

1.8.1 Data Structure Descriptions

Supported processors are “little endian” machines. This distinction means that the low-order byte of a multibyte data item in memory is at the lowest address, while the high-order byte is at the highest address. Some supported 64-bit processors may be configured for both “little endian” and “big endian” operation. All implementations designed to conform to this specification use “little endian” operation.

In some memory layout descriptions, certain fields are marked reserved. Software must initialize such fields to zero and ignore them when read. On an update operation, software must preserve any reserved field.

1.8.2 Protocol Descriptions

A protocol description generally has the following format:

Protocol Name: The formal name of the protocol interface.

Summary: A brief description of the protocol interface.

GUID: The 128-bit Globally Unique Identifier (GUID) for the protocol interface.

Protocol Interface Structure: A “C-style” data structure definition containing the procedures and data fields produced by this protocol interface.

Parameters: A brief description of each field in the protocol interface structure.

Description: A description of the functionality provided by the interface, including any limitations and caveats of which the caller should be aware.

Related Definitions: The type declarations and constants that are used in the protocol interface structure or any of its procedures.

1.8.3 Procedure Descriptions

A procedure description generally has the following format:

ProcedureName(): The formal name of the procedure.

Summary: A brief description of the procedure.

Prototype: A “C-style” procedure header defining the calling sequence.
Parameters: A brief description of each field in the procedure prototype.

Description: A description of the functionality provided by the interface, including any limitations and caveats of which the caller should be aware.

Related Definitions: The type declarations and constants that are used only by this procedure.

Status Codes Returned: A description of any codes returned by the interface. The procedure is required to implement any status codes listed in this table. Additional error codes may be returned, but they will not be tested by standard compliance tests, and any software that uses the procedure cannot depend on any of the extended error codes that an implementation may provide.

1.8.4 Instruction Descriptions
An instruction description for EBC instructions generally has the following format:

InstructionName: The formal name of the instruction.

Syntax: A brief description of the instruction.

Description: A description of the functionality provided by the instruction accompanied by a table that details the instruction encoding.

Operation: Details the operations performed on operands.

Behaviors and Restrictions: An item-by-item description of the behavior of each operand involved in the instruction and any restrictions that apply to the operands or the instruction.

1.8.5 Pseudo-Code Conventions
Pseudo code is presented to describe algorithms in a more concise form. None of the algorithms in this document are intended to be compiled directly. The code is presented at a level corresponding to the surrounding text.

In describing variables, a list is an unordered collection of homogeneous objects. A queue is an ordered list of homogeneous objects. Unless otherwise noted, the ordering is assumed to be FIFO.

Pseudo code is presented in a C-like format, using C conventions where appropriate. The coding style, particularly the indentation style, is used for readability and does not necessarily comply with an implementation of the UEFI Specification.

1.8.6 Typographic Conventions
This document uses the typographic and illustrative conventions described below:

- **Plain text** The normal text typeface is used for the vast majority of the descriptive text in a specification.

- **Plain text (blue)** Any plain text that is underlined and in blue indicates an active link to the cross-reference. Click on the word to follow the hyperlink.

- **Bold** In text, a Bold typeface identifies a processor register name. In other instances, a Bold typeface can be used as a running head within a paragraph.

- **Italic** In text, an Italic typeface can be used as emphasis to introduce a new term or to indicate a manual or specification name.
**BOLD Monospace** Computer code, example code segments, and all prototype code segments use a **BOLD Monospace** typeface with a dark red color. These code listings normally appear in one or more separate paragraphs, though words or segments can also be embedded in a normal text paragraph.

**Bold Monospace** Words in a **Bold Monospace** typeface that is underlined and in blue indicate an active hyperlink to the code definition for that function or type definition. Click on the word to follow the hyperlink.

**Note:** Due to management and file size considerations, only the first occurrence of the reference on each page is an active link. Subsequent references on the same page will not be actively linked to the definition and will use the standard, nonunderlined **BOLD Monospace** typeface. Find the first instance of the name (in the underlined **BOLD Monospace** typeface) on the page and click on the word to jump to the function or type definition.

**Italic Monospace** In code or in text, words in **Italic Monospace** indicate placeholder names for variable information that must be supplied (i.e., arguments).

### 1.8.7 Number formats

A binary number is represented in this standard by any sequence of digits consisting of only the Western-Arabic numerals 0 and 1 immediately followed by a lower-case b (e.g., 0101b).

Underscores or spaces may be included between characters in binary number representations to increase readability or delineate field boundaries (e.g., 0 0101 1010b or 0_0101_1010b).

#### 1.8.7.1 Hexadecimal

A hexadecimal number is represented in this standard by 0x preceding any sequence of digits consisting of only the Western-Arabic numerals 0 through 9 and/or the upper-case English letters A through F (e.g., 0xFA23).

Underscores or spaces may be included between characters in hexadecimal number representations to increase readability or delineate field boundaries (e.g., 0xB_FD8C_FA23 or 0xB_FD8C_FA23).

#### 1.8.7.2 Decimal

A decimal number is represented in this standard by any sequence of digits consisting of only the Arabic numerals 0 through 9 not immediately followed by a lower-case b or lower-case h (e.g., 25).

This standard uses the following conventions for representing decimal numbers:

- the decimal separator (i.e., separating the integer and fractional portions of the number) is a period;
- the thousands separator (i.e., separating groups of three digits in a portion of the number) is a comma;
- the thousands separator is used in the integer portion and is not used in the fraction portion of a number.
1.8.8 Binary prefixes

This standard uses the prefixes defined in the International System of Units (SI) for values that are powers of ten. See “Links to UEFI-Related Documents” (http://uefi.org/uefi) under the heading "SI Binary Prefixes”.

<table>
<thead>
<tr>
<th>Factor</th>
<th>Name</th>
<th>Symbol</th>
</tr>
</thead>
<tbody>
<tr>
<td>10³</td>
<td>1,000</td>
<td>kilo</td>
</tr>
<tr>
<td>10⁶</td>
<td>1,000,000</td>
<td>mega</td>
</tr>
<tr>
<td>10⁹</td>
<td>1,000,000,000</td>
<td>giga</td>
</tr>
</tbody>
</table>

This standard uses the binary prefixes defined in ISO/IEC 80000-13 Quantities and units -- Part 13: Information science and technology and IEEE 1514 Standard for Prefixes for Binary Multiples for values that are powers of two.

<table>
<thead>
<tr>
<th>Factor</th>
<th>Factor</th>
<th>Name</th>
<th>Symbol</th>
</tr>
</thead>
<tbody>
<tr>
<td>2¹⁰</td>
<td>1,024</td>
<td>kibi</td>
<td>Ki</td>
</tr>
<tr>
<td>2²⁰</td>
<td>1,048,576</td>
<td>mebi</td>
<td>Mi</td>
</tr>
<tr>
<td>2³⁰</td>
<td>1,073,741,824</td>
<td>gibi</td>
<td>Gi</td>
</tr>
</tbody>
</table>

For example, 4 KB means 4,000 bytes and 4 KiB means 4,096 bytes.

1.8.9 Revision Numbers

Updates to the UEFI specification are considered either new revisions or errata as described below:

- A new revision is produced when there is substantive new content or changes that may modify existing behavior. New revisions are designated by a major.minor version number (e.g. xx.yy). In cases where the changes are exceptionally minor, we may have a major.minor.minor naming convention (e.g. xx.yy.zz).
- Errata versions are produced when approved updates to the specification do not include any significant new material or modify existing behavior. Errata are designated by adding an upper-case letter at the end of the version number, such as xx.yy errata A.
2 - Overview

UEFI allows the extension of platform firmware by loading UEFI driver and UEFI application images. When UEFI drivers and UEFI applications are loaded they have access to all UEFI-defined runtime and boot services. See Figure 2-1.

![Figure 2-1 Booting Sequence](image)

UEFI allows the consolidation of boot menus from the OS loader and platform firmware into a single platform firmware menu. These platform firmware menus will allow the selection of any UEFI OS loader from any partition on any boot medium that is supported by UEFI boot services. An UEFI OS loader can support multiple options that can appear on the user interface. It is also possible to include legacy boot options, such as booting from the A: or C: drive in the platform firmware boot menus.

UEFI supports booting from media that contain an UEFI OS loader or an UEFI-defined System Partition. An UEFI-defined System Partition is required by UEFI to boot from a block device. UEFI does not require any change to the first sector of a partition, so it is possible to build media that will boot on both legacy architectures and UEFI platforms.

2.1 Boot Manager

UEFI contains a boot manager that allows the loading of applications written to this specification (including OS 1st stage loader) or UEFI drivers from any file on an UEFI-defined file system or through the use of an UEFI-defined image loading service. UEFI defines NVRAM variables that are used to point to the file to be loaded. These variables also contain application-specific data that are passed directly to the UEFI application. The variables also contain a human readable string that can be displayed in a menu to the user.
The variables defined by UEFI allow the system firmware to contain a boot menu that can point to all of the operating systems, and even multiple versions of the same operating systems. The design goal of UEFI was to have one set of boot menus that could live in platform firmware. UEFI specifies only the NVRAM variables used in selecting boot options. UEFI leaves the implementation of the menu system as value added implementation space.

UEFI greatly extends the boot flexibility of a system over the current state of the art in the PC-AT-class system. The PC-AT-class systems today are restricted to boot from the first floppy, hard drive, CD-ROM, USB keys, or network card attached to the system. Booting from a common hard drive can cause many interoperability problems between operating systems, and different versions of operating systems from the same vendor.

2.1.1 UEFI Images

UEFI Images are a class of files defined by UEFI that contain executable code. The most distinguishing feature of UEFI Images is that the first set of bytes in the UEFI Image file contains an image header that defines the encoding of the executable image.

UEFI uses a subset of the PE32+ image format with a modified header signature. The modification to the signature value in the PE32+ image is done to distinguish UEFI images from normal PE32 executables. The “+” addition to PE32 provides the 64-bit relocation fix-up extensions to standard PE32 format.

For images with the UEFI image signature, the Subsystem values in the PE image header are defined below. The major differences between image types are the memory type that the firmware will load the image into, and the action taken when the image’s entry point exits or returns. A UEFI application image is always unloaded when control is returned from the image’s entry point. A UEFI driver image is only unloaded if control is passed back with a UEFI error code.

```
// PE32+ Subsystem type for EFI images
#define EFI_IMAGE_SUBSYSTEM_EFI_APPLICATION 10
#define EFI_IMAGE_SUBSYSTEM_EFI_BOOT_SERVICE_DRIVER 11
#define EFI_IMAGE_SUBSYSTEM_EFI_RUNTIME_DRIVER 12

// PE32+ Machine type for EFI images
#define EFI_IMAGE_MACHINE_IA32 0x014c
#define EFI_IMAGE_MACHINE_IA64 0x0200
#define EFI_IMAGE_MACHINE_EBC 0x0EBC
#define EFI_IMAGE_MACHINE_x64 0x8664
#define EFI_IMAGE_MACHINE_ARMTHUMB_MIXED 0x01C2
#define EFI_IMAGE_MACHINE_AARCH64 0xAA64
#define EFI_IMAGE_MACHINE_RISCV32 0x5032
#define EFI_IMAGE_MACHINE_RISCV64 0x5064
#define EFI_IMAGE_MACHINE_RISCV128 0x5128
```

**Note:** This image type is chosen to enable UEFI images to contain Thumb and Thumb2 instructions while defining the EFI interfaces themselves to be in ARM mode.
The **Machine** value that is found in the PE image file header is used to indicate the machine code type of the image. The machine code types for images with the UEFI image signature are defined below. A given platform must implement the image type native to that platform and the image type for EFI Byte Code (EBC). Support for other machine code types is optional to the platform.

A UEFI image is loaded into memory through the **EFI_BOOT_SERVICES.LoadImage()** Boot Service. This service loads an image with a PE32+ format into memory. This PE32+ loader is required to load all sections of the PE32+ image into memory. Once the image is loaded into memory, and the appropriate fix-ups have been performed, control is transferred to a loaded image at the **AddressOfEntryPoint** reference according to the normal indirect calling conventions of applications based on supported 32-bit, 64-bit, or 128-bit processors. All other linkage to and from an UEFI image is done programmatically.

### 2.1.2 UEFI Applications

Applications written to this specification are loaded by the Boot Manager or by other UEFI applications. To load a UEFI application the firmware allocates enough memory to hold the image, copies the sections within the UEFI application image to the allocated memory, and applies the relocation fix-ups needed. Once done, the allocated memory is set to be the proper type for code and data for the image. Control is then transferred to the UEFI application’s entry point. When the application returns from its entry point, or when it calls the Boot Service **EFI_BOOT_SERVICES.Exit()**, the UEFI application is unloaded from memory and control is returned to the UEFI component that loaded the UEFI application.

When the Boot Manager loads a UEFI application, the image handle may be used to locate the “load options” for the UEFI application. The load options are stored in nonvolatile storage and are associated with the UEFI application being loaded and executed by the Boot Manager.

### 2.1.3 UEFI OS Loaders

A UEFI OS loader is a special type of UEFI application that normally takes over control of the system from firmware conforming to this specification. When loaded, the UEFI OS loader behaves like any other UEFI application in that it must only use memory it has allocated from the firmware and can only use UEFI services and protocols to access the devices that the firmware exposes. If the UEFI OS loader includes any boot service style driver functions, it must use the proper UEFI interfaces to obtain access to the bus specific-resources. That is, I/O and memory-mapped device registers must be accessed through the proper bus specific I/O calls like those that a UEFI driver would perform.

If the UEFI OS loader experiences a problem and cannot load its operating system correctly, it can release all allocated resources and return control back to the firmware via the Boot Service **Exit()** call. The **Exit()** call allows both an error code and **ExitData** to be returned. The **ExitData** contains both a string and OS loader-specific data to be returned.

---

**Table 2-1 UEFI Image Memory Types**

<table>
<thead>
<tr>
<th>Subsystem Type</th>
<th>Code Memory Type</th>
<th>Data Memory Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>EFI_IMAGE_SUBSYSTEM_EFI_APPLICATION</td>
<td>EfiLoaderCode</td>
<td>EfiLoaderData</td>
</tr>
<tr>
<td>EFI_IMAGE_SUBSYSTEM_EFI_BOOT_SERVICE_DRIVER</td>
<td>EfiBootServicesCode</td>
<td>EfiBootServicesData</td>
</tr>
<tr>
<td>EFI_IMAGE_SUBSYSTEM_EFI_RUNTIME_DRIVER</td>
<td>EfiRuntimeServicesCode</td>
<td>EfiRuntimeServicesData</td>
</tr>
</tbody>
</table>

---
If the UEFI OS loader successfully loads its operating system, it can take control of the system by using the Boot Service `EFI_BOOT_SERVICES.ExitBootServices()`. After successfully calling `ExitBootServices()`, all boot services in the system are terminated, including memory management, and the UEFI OS loader is responsible for the continued operation of the system.

### 2.1.4 UEFI Drivers

UEFI drivers are loaded by the Boot Manager, firmware conforming to this specification, or by other UEFI applications. To load a UEFI driver the firmware allocates enough memory to hold the image, copies the sections within the UEFI driver image to the allocated memory and applies the relocation fix-ups needed. Once done, the allocated memory is set to be the proper type for code and data for the image. Control is then transferred to the UEFI driver’s entry point. When the UEFI driver returns from its entry point, or when it calls the Boot Service `EFI_BOOT_SERVICES.Exit()`, the UEFI driver is optionally unloaded from memory and control is returned to the component that loaded the UEFI driver. A UEFI driver is not unloaded from memory if it returns a status code of `EFI_SUCCESS`. If the UEFI driver’s return code is an error status code, then the driver is unloaded from memory.

There are two types of UEFI drivers: boot service drivers and runtime drivers. The only difference between these two driver types is that UEFI runtime drivers are available after a UEFI OS loader has taken control of the platform with the Boot Service `EFI_BOOT_SERVICES.ExitBootServices()`.

UEFI boot service drivers are terminated when `ExitBootServices()` is called, and all the memory resources consumed by the UEFI boot service drivers are released for use in the operating system environment.

A runtime driver of type `EFI_IMAGE_SUBSYSTEM_EFI_RUNTIME_DRIVER` gets fixed up with virtual mappings when the OS calls `SetVirtualAddressMap()`.

### 2.2 Firmware Core

This section provides an overview of the services defined by UEFI. These include boot services and runtime services.

#### 2.2.1 UEFI Services

The purpose of the UEFI interfaces is to define a common boot environment abstraction for use by loaded UEFI images, which include UEFI drivers, UEFI applications, and UEFI OS loaders. The calls are defined with a full 64-bit interface, so that there is headroom for future growth. The goal of this set of abstracted platform calls is to allow the platform and OS to evolve and innovate independently of one another. Also, a standard set of primitive runtime services may be used by operating systems.

Platform interfaces defined in this section allow the use of standard Plug and Play Option ROMs as the underlying implementation methodology for the boot services. The interfaces have been designed in such as way as to map back into legacy interfaces. These interfaces have in no way been burdened with any restrictions inherent to legacy Option ROMs.

The UEFI platform interfaces are intended to provide an abstraction between the platform and the OS that is to boot on the platform. The UEFI specification also provides abstraction between diagnostics or utility programs and the platform; however, it does not attempt to implement a full diagnostic OS environment. It is envisioned that a small diagnostic OS-like environment can be easily built on top of an UEFI system. Such a diagnostic environment is not described by this specification.
Interfaces added by this specification are divided into the following categories and are detailed later in this document:

- Runtime services
- Boot services interfaces, with the following subcategories:
  - Global boot service interfaces
  - Device handle-based boot service interfaces
  - Device protocols
  - Protocol services

### 2.2.2 Runtime Services

This section describes UEFI runtime service functions. The primary purpose of the runtime services is to abstract minor parts of the hardware implementation of the platform from the OS. Runtime service functions are available during the boot process and also at runtime provided the OS switches into flat physical addressing mode to make the runtime call. However, if the OS loader or OS uses the Runtime Service `SetVirtualAddressMap()` service, the OS will only be able to call runtime services in a virtual addressing mode. All runtime interfaces are non-blocking interfaces and can be called with interrupts disabled if desired. To ensure maximum compatibility with existing platforms it is recommended that all UEFI modules that comprise the Runtime Services be represented in the `MemoryMap` as a single `EFI_MEMORY_DESCRIPTOR` of Type `EfiRuntimeServicesCode`.

In all cases memory used by the runtime services must be reserved and not used by the OS. Runtime services memory is always available to an UEFI function and will never be directly manipulated by the OS or its components. UEFI is responsible for defining the hardware resources used by runtime services, so the OS can synchronize with those resources when runtime service calls are made, or guarantee that the OS never uses those resources.

Table 2-2 lists the Runtime Services functions.

<table>
<thead>
<tr>
<th>Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>GetTime()</code></td>
<td>Returns the current time, time context, and time keeping capabilities.</td>
</tr>
<tr>
<td><code>SetTime()</code></td>
<td>Sets the current time and time context.</td>
</tr>
<tr>
<td><code>GetWakeupTime()</code></td>
<td>Returns the current wakeup alarm settings.</td>
</tr>
<tr>
<td><code>SetWakeupTime()</code></td>
<td>Sets the current wakeup alarm settings.</td>
</tr>
<tr>
<td><code>GetVariable()</code></td>
<td>Returns the value of a named variable.</td>
</tr>
<tr>
<td><code>GetNextVariableName()</code></td>
<td>Enumerates variable names.</td>
</tr>
<tr>
<td><code>SetVariable()</code></td>
<td>Sets, and if needed creates, a variable.</td>
</tr>
<tr>
<td><code>SetVirtualAddressMap()</code></td>
<td>Switches all runtime functions from physical to virtual addressing.</td>
</tr>
<tr>
<td><code>ConvertPointer()</code></td>
<td>Used to convert a pointer from physical to virtual addressing.</td>
</tr>
</tbody>
</table>
2.3 Calling Conventions

Unless otherwise stated, all functions defined in the UEFI specification are called through pointers in common, architecturally defined, calling conventions found in C compilers. Pointers to the various global UEFI functions are found in the EFI_RUNTIME_SERVICES and EFI_BOOT_SERVICES tables that are located via the system table. Pointers to other functions defined in this specification are located dynamically through device handles. In all cases, all pointers to UEFI functions are cast with the word EFIAPI. This allows the compiler for each architecture to supply the proper compiler keywords to achieve the needed calling conventions. When passing pointer arguments to Boot Services, Runtime Services, and Protocol Interfaces, the caller has the following responsibilities:

- It is the caller’s responsibility to pass pointer parameters that reference physical memory locations. If a pointer is passed that does not point to a physical memory location (i.e., a memory mapped I/O region), the results are unpredictable and the system may halt.
- It is the caller’s responsibility to pass pointer parameters with correct alignment. If an unaligned pointer is passed to a function, the results are unpredictable and the system may halt.
- It is the caller’s responsibility to not pass in a NULL parameter to a function unless it is explicitly allowed. If a NULL pointer is passed to a function, the results are unpredictable and the system may hang.
- Unless otherwise stated, a caller should not make any assumptions regarding the state of pointer parameters if the function returns with an error.
- A caller may not pass structures that are larger than native size by value and these structures must be passed by reference (via a pointer) by the caller. Passing a structure larger than native width (4 bytes on supported 32-bit processors; 8 bytes on supported 64-bit processor instructions) on the stack will produce undefined results.

Calling conventions for supported 32-bit and supported 64-bit applications are described in more detail below. Any function or protocol may return any valid return code.

All public interfaces of a UEFI module must follow the UEFI calling convention. Public interfaces include the image entry point, UEFI event handlers, and protocol member functions. The type EFIAPI is used to indicate conformance to the calling conventions defined in this section. Non public interfaces, such as private functions and static library calls, are not required to follow the UEFI calling conventions and may be optimized by the compiler.
2.3.1 Data Types

Table 2-3 lists the common data types that are used in the interface definitions, and Table 2-4 lists their modifiers. Unless otherwise specified all data types are naturally aligned. Structures are aligned on boundaries equal to the largest internal datum of the structure and internal data are implicitly padded to achieve natural alignment.

The values of the pointers passed into or returned by the UEFI interfaces must provide natural alignment for the underlying types.

<table>
<thead>
<tr>
<th>Mnemonic</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>BOOLEAN</td>
<td>Logical Boolean. 1-byte value containing a 0 for FALSE or a 1 for TRUE. Other values are undefined.</td>
</tr>
<tr>
<td>INTN</td>
<td>Signed value of native width. (4 bytes on supported 32-bit processor instructions, 8 bytes on supported 64-bit processor instructions, 16 bytes on supported 128-bit processor instructions)</td>
</tr>
<tr>
<td>UINTN</td>
<td>Unsigned value of native width. (4 bytes on supported 32-bit processor instructions, 8 bytes on supported 64-bit processor instructions, 16 bytes on supported 128-bit processor instructions)</td>
</tr>
<tr>
<td>INT8</td>
<td>1-byte signed value.</td>
</tr>
<tr>
<td>UINT8</td>
<td>1-byte unsigned value.</td>
</tr>
<tr>
<td>INT16</td>
<td>2-byte signed value.</td>
</tr>
<tr>
<td>UINT16</td>
<td>2-byte unsigned value.</td>
</tr>
<tr>
<td>INT32</td>
<td>4-byte signed value.</td>
</tr>
<tr>
<td>UINT32</td>
<td>4-byte unsigned value.</td>
</tr>
<tr>
<td>INT64</td>
<td>8-byte signed value.</td>
</tr>
<tr>
<td>UINT64</td>
<td>8-byte unsigned value.</td>
</tr>
<tr>
<td>INT128</td>
<td>16-byte signed value.</td>
</tr>
<tr>
<td>UINT128</td>
<td>16-byte unsigned value.</td>
</tr>
<tr>
<td>CHAR8</td>
<td>1-byte character. Unless otherwise specified, all 1-byte or ASCII characters and strings are stored in 8-bit ASCII encoding format, using the ISO-Latin-1 character set.</td>
</tr>
<tr>
<td>CHAR16</td>
<td>2-byte Character. Unless otherwise specified all characters and strings are stored in the UCS-2 encoding format as defined by Unicode 2.1 and ISO/IEC 10646 standards.</td>
</tr>
<tr>
<td>VOID</td>
<td>Undeclared type.</td>
</tr>
<tr>
<td>EFI_GUID</td>
<td>128-bit buffer containing a unique identifier value. Unless otherwise specified, aligned on a 64-bit boundary.</td>
</tr>
<tr>
<td>EFI_STATUS</td>
<td>Status code. Type UINTN.</td>
</tr>
<tr>
<td>EFI_HANDLE</td>
<td>A collection of related interfaces. Type VOID *.</td>
</tr>
<tr>
<td>EFI_EVENT</td>
<td>Handle to an event structure. Type VOID *.</td>
</tr>
<tr>
<td>EFI_LBA</td>
<td>Logical block address. Type UINT64.</td>
</tr>
<tr>
<td>EFI_TPL</td>
<td>Task priority level. Type UINTN.</td>
</tr>
</tbody>
</table>
2.3.2 IA-32 Platforms

All functions are called with the C language calling convention. The general-purpose registers that are volatile across function calls are `eax`, `ecx`, and `edx`. All other general-purpose registers are nonvolatile and are preserved by the target function. In addition, unless otherwise specified by the function definition, all other registers are preserved.

Firmware boot services and runtime services run in the following processor execution mode prior to the OS calling ExitBootServices():

- Uniprocessor, as described in chapter 8.4 of:
  - Intel 64 and IA-32 Architectures Software Developer's Manual
  - Volume 3, System Programming Guide, Part 1
  - Order Number: 253668-033US, December 2009
  - See “Links to UEFI-Related Documents” (http://uefi.org/uefi) under the heading "Intel Processor Manuals.

- Protected mode

---

**Table 2-4 Modifiers for Common UEFI Data Types**

<table>
<thead>
<tr>
<th>Mnemonic</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>EFIMAC_ADDRESS</strong></td>
<td>32-byte buffer containing a network Media Access Control address.</td>
</tr>
<tr>
<td><strong>EFI_IPV4_ADDRESS</strong></td>
<td>4-byte buffer. An IPv4 internet protocol address.</td>
</tr>
<tr>
<td><strong>EFI_IPV6_ADDRESS</strong></td>
<td>16-byte buffer. An IPv6 internet protocol address.</td>
</tr>
<tr>
<td><strong>EFI_IP_ADDRESS</strong></td>
<td>16-byte buffer aligned on a 4-byte boundary. An IPv4 or IPv6 internet protocol address.</td>
</tr>
<tr>
<td><code>&lt;Enumerated Type&gt;</code></td>
<td>Element of a standard ANSI C <code>enum</code> type declaration. Type INT32 or UINT32. ANSI C does not define the size of sign of an enum so they should never be used in structures. ANSI C integer promotion rules make INT32 or UINT32 interchangeable when passed as an argument to a function.</td>
</tr>
<tr>
<td><code>sizeof (VOID *)</code></td>
<td>4 bytes on supported 32-bit processor instructions. 8 bytes on supported 64-bit processor instructions. 16 bytes on supported 128-bit processor.</td>
</tr>
</tbody>
</table>

**Bitfields**

Bitfields are ordered such that bit 0 is the least significant bit.
• Paging mode may be enabled. If paging mode is enabled, PAE (Physical Address Extensions) mode is recommended. If paging mode is enabled, any memory space defined by the UEFI memory map is identity mapped (virtual address equals physical address). The mappings to other regions are undefined and may vary from implementation to implementation.
• Selectors are set to be flat and are otherwise not used
• Interrupts are enabled—though no interrupt services are supported other than the UEFI boot services timer functions (All loaded device drivers are serviced synchronously by “polling.”)
• Direction flag in EFLAGS is clear
• Other general purpose flag registers are undefined
• 128 KiB, or more, of available stack space
• The stack must be 16-byte aligned. Stack may be marked as non-executable in identity mapped page tables.
• Floating-point control word must be initialized to 0x027F (all exceptions masked, double-precision, round-to-nearest)
• Multimedia-extensions control word (if supported) must be initialized to 0x1F80 (all exceptions masked, round-to-nearest, flush to zero for masked underflow).
• CR0.EM must be zero
• CR0.TS must be zero

An application written to this specification may alter the processor execution mode, but the UEFI image must ensure firmware boot services and runtime services are executed with the prescribed execution environment.

After an Operating System calls `ExitBootServices()`, firmware boot services are no longer available and it is illegal to call any boot service. After ExitBootServices, firmware runtime services are still available and may be called with paging enabled and virtual address pointers if `SetVirtualAddressMap()` has been called describing all virtual address ranges used by the firmware runtime service.

For an operating system to use any UEFI runtime services, it must:

• Preserve all memory in the memory map marked as runtime code and runtime data
• Call the runtime service functions, with the following conditions:
  — In protected mode
  — Paging may or may not be enabled, however if paging is enabled and `SetVirtualAddressMap()` has not been called, any memory space defined by the UEFI memory map is identity mapped (virtual address equals physical address), although the attributes of certain regions may not have all read, write, and execute attributes or be unmarked for purposes of platform protection. The mappings to other regions are undefined and may vary from implementation to implementation. See description of `SetVirtualAddressMap()` for details of memory map after this function has been called.
    — Direction flag in EFLAGS clear
    — 4 KiB, or more, of available stack space
    — The stack must be 16-byte aligned
Floating-point control word must be initialized to 0x027F (all exceptions masked, double-precision, round-to-nearest)

Multimedia-extensions control word (if supported) must be initialized to 0x1F80 (all exceptions masked, round-to-nearest, flush to zero for masked underflow)

CR0.EM must be zero

CR0.TS must be zero

Interrupts disabled or enabled at the discretion of the caller

- ACPI Tables loaded at boot time can be contained in memory of type EfiACPIReclaimMemory (recommended) or EfiACPIMemoryNVS. ACPI FACS must be contained in memory of type EfiACPIMemoryNVS.

- The system firmware must not request a virtual mapping for any memory descriptor of type EfiACPIReclaimMemory or EfiACPIMemoryNVS.

- EFI memory descriptors of type EfiACPIReclaimMemory and EfiACPIMemoryNVS must be aligned on a 4 KiB boundary and must be a multiple of 4 KiB in size.

- Any UEFI memory descriptor that requests a virtual mapping via the EFI MEMORY_DESCRIPTOR having the EFI_MEMORY_RUNTIME bit set must be aligned on a 4 KiB boundary and must be a multiple of 4 KiB in size.

- An ACPI Memory Op-region must inherit cacheability attributes from the UEFI memory map. If the system memory map does not contain cacheability attributes, the ACPI Memory Op-region must inherit its cacheability attributes from the ACPI name space. If no cacheability attributes exist in the system memory map or the ACPI name space, then the region must be assumed to be non-cacheable.

- ACPI tables loaded at runtime must be contained in memory of type EfiACPIMemoryNVS. The cacheability attributes for ACPI tables loaded at runtime should be defined in the UEFI memory map. If no information about the table location exists in the UEFI memory map, cacheability attributes may be obtained from ACPI memory descriptors. If no information about the table location exists in the UEFI memory map or ACPI memory descriptors, the table is assumed to be non-cached.

- In general, UEFI Configuration Tables loaded at boot time (e.g., SMBIOS table) can be contained in memory of type EfiRuntimeServicesData (recommended), EfiBootServicesData, EfiACPIReclaimMemory or EfiACPIMemoryNVS. Tables loaded at runtime must be contained in memory of type EfiRuntimeServicesData (recommended) or EfiACPIMemoryNVS.

Note: Previous EFI specifications allowed ACPI tables loaded at runtime to be in the EfiReservedMemoryType and there was no guidance provided for other EFI Configuration Tables. EfiReservedMemoryType is not intended to be used for the storage of any EFI Configuration Tables. Also, only OSes conforming to the UEFI Specification are guaranteed to handle SMBIOS table in memory of type EfiBootServicesData.

2.3.2.1 Handoff State

When a 32-bit UEFI OS is loaded, the system firmware hands off control to the OS in flat 32-bit mode. All descriptors are set to their 4GiB limits so that all of memory is accessible from all segments.
Figure 2-2 shows the stack after AddressOfEntryPoint in the image’s PE32+ header has been called on supported 32-bit systems. All UEFI image entry points take two parameters. These are the image handle of the UEFI image, and a pointer to the EFI System Table.

<table>
<thead>
<tr>
<th>Stack</th>
<th>Location</th>
</tr>
</thead>
<tbody>
<tr>
<td>EFI_SYSTEM_TABLE *</td>
<td>ESP + 8</td>
</tr>
<tr>
<td>EFI_HANDLE</td>
<td>ESP + 4</td>
</tr>
<tr>
<td>&lt;return address&gt;</td>
<td>ESP</td>
</tr>
</tbody>
</table>

Figure 2-2 Stack after AddressOfEntryPoint Called, IA-32

2.3.2.2 Calling Convention

All functions are called with the C language calling convention. The general-purpose registers that are volatile across function calls are eax, ecx, and edx. All other general-purpose registers are nonvolatile and are preserved by the target function.

In addition, unless otherwise specified by the function definition, all other CPU registers (including MMX and XMM) are preserved.

The floating point status register is not preserved by the target function. The floating point control register and MMX control register are saved by the target function.

If the return value is a float or a double, the value is returned in ST(0).

2.3.3 Intel® Itanium®-Based Platforms

UEFI executes as an extension to the SAL execution environment with the same rules as laid out by the SAL specification.

During boot services time the processor is in the following execution mode:

- Uniprocessor, as detailed in chapter 13.1.2 of:
  - Intel Itanium Architecture Software Developer’s Manual
  - Volume 2: System Architecture
  - Revision 2.2
  - January 2006
  - See “Links to UEFI-Related Documents” (http://uefi.org/uefi) under the heading “Intel Itanium Documentation”.
  - Document Number: 245318-005
- Physical mode
- 128 KiB, or more, of available stack space
- 16 KiB, or more, of available backing store space
  - FPSR.traps: Set to all 1’s (all exceptions disabled)
  - FPSR.sf0:
• .pc: Precision Control - 11b (extended precision)
• .rc: Rounding Control - 0 (round to nearest)
• .wre: Widest Range Exponent - 0 (IEEE mode)
• .ftz: Flush-To-Zero mode - 0 (off)
  — FPSR.sf1:
• .td: Traps Disable = 1 (traps disabled)
• .pc: Precision Control - 11b (extended precision)
• .rc: Rounding Control - 0 (round to nearest)
• .wre: Widest Range Exponent - 1 (full register exponent range)
• .ftz: Flush-To-Zero mode - 0 (off)
  — FPSR.sf2,3:
• .td: Traps Disable = 1 (traps disabled)
• .pc: Precision Control - 11b (extended precision)
• .rc: Rounding Control - 0 (round to nearest)
• .wre: Widest Range Exponent - 0 (IEEE mode)
• .ftz: Flush-To-Zero mode - 0 (off)

An application written to this specification may alter the processor execution mode, but the UEFI image must ensure firmware boot services and runtime services are executed with the prescribed execution environment.

After an Operating System calls ExitBootServices(), firmware boot services are no longer available and it is illegal to call any boot service. After ExitBootServices(), firmware runtime services are still available. When calling runtime services, paging may or may not be enabled, however if paging is enabled and SetVirtualAddressMap() has not been called, any memory space defined by the UEFI memory map is identity mapped (virtual address equals physical address). The mappings to other regions are undefined and may vary from implementation to implementation. See description of SetVirtualAddressMap() for details of memory map after this function has been called. After ExitBootServices(), runtime service functions may be called with interrupts disabled or enabled at the discretion of the caller.

• ACPI Tables loaded at boot time can be contained in memory of type EfiACPIReclaimMemory (recommended) or EfiACPIMemoryNV. ACPI FACS must be contained in memory of type EfiACPIMemoryNV.

• The system firmware must not request a virtual mapping for any memory descriptor of type EfiACPIReclaimMemory or EfiACPIMemoryNV.

• EFI memory descriptors of type EfiACPIReclaimMemory and EfiACPIMemoryNV must be aligned on an 8 KiB boundary and must be a multiple of 8 KiB in size.

• Any UEFI memory descriptor that requests a virtual mapping via the EFI_MEMORY_DESCRIPTOR having the EFI_MEMORY_RUNTIME bit set must be aligned on an 8 KiB boundary and must be a multiple of 8 KiB in size.

• An ACPI Memory Op-region must inherit cacheability attributes from the UEFI memory map. If the system memory map does not contain cacheability attributes the ACPI Memory Op-region must inherit its cacheability attributes from the ACPI name space. If no cacheability attributes
exist in the system memory map or the ACPI name space, then the region must be assumed to be non-cacheable.

- ACPI tables loaded at runtime must be contained in memory of type `EfiACPIMemoryNVS`. The cacheability attributes for ACPI tables loaded at runtime should be defined in the UEFI memory map. If no information about the table location exists in the UEFI memory map, cacheability attributes may be obtained from ACPI memory descriptors. If no information about the table location exists in the UEFI memory map or ACPI memory descriptors, the table is assumed to be non-cached.

- In general, Configuration Tables loaded at boot time (e.g., SMBIOS table) can be contained in memory of type `EfiRuntimeServicesData` (recommended), `EfiBootServicesData`, `EfiACPIReclaimMemory` or `EfiACPIMemoryNVS`. Tables loaded at runtime must be contained in memory of type `EfiRuntimeServicesData` (recommended) or `EfiACPIMemoryNVS`.

**Note:** Previous EFI specifications allowed ACPI tables loaded at runtime to be in the `EfiReservedMemoryType` and there was no guidance provided for other EFI Configuration Tables. `EfiReservedMemoryType` is not intended to be used by firmware. Also, only OSes conforming to the UEFI Specification are guaranteed to handle SMBIOS table in memory of type `EfiBootServicesData`.

Refer to the IA-64 System Abstraction Layer Specification (see Appendix Q) for details.

UEFI procedures are invoked using the P64 C calling conventions defined for Intel® Itanium®-based applications. Refer to the document 64 Bit Runtime Architecture and Software Conventions for IA-64 (see Appendix Q) for more information.

### 2.3.3.1 Handoff State

UEFI uses the standard P64 C calling conventions that are defined for Itanium-based operating systems. Figure 2-3 shows the stack after `ImageEntryPoint` has been called on Itanium-based systems. The arguments are also stored in registers: `out0` contains `EFI_HANDLE` and `out1` contains the address of the `EFI_SYSTEM_TABLE`. The `gp` for the UEFI Image will have been loaded from the `plabel` pointed to by the `AddressOfEntryPoint` in the image’s PE32+ header. All UEFI image entry points take two parameters. These are the image handle of the image, and a pointer to the System Table.

<table>
<thead>
<tr>
<th>Stack</th>
<th>Location</th>
<th>Register</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>EFI_SYSTEM_TABLE</code> *</td>
<td>SP + 8</td>
<td>out1</td>
</tr>
<tr>
<td><code>EFI_HANDLE</code></td>
<td>SP</td>
<td>out0</td>
</tr>
</tbody>
</table>

Figure 2-3 Stack after `AddressOfEntryPoint` Called, Itanium-based Systems

The SAL specification (see Appendix Q) defines the state of the system registers at boot handoff. The SAL specification also defines which system registers can only be used after UEFI boot services have been properly terminated.
2.3.3.2 Calling Convention

UEFI executes as an extension to the SAL execution environment with the same rules as laid out by the SAL specification. UEFI procedures are invoked using the P64 C calling conventions defined for Intel® Itanium®-based applications. Refer to the document 64 Bit Runtime Architecture and Software Conventions for IA-64 (see Glossary for more information).

For floating point, functions may only use the lower 32 floating point registers. Return values appear in f8-f15 registers. Single, double, and extended values are all returned using the appropriate format. Registers f6-f7 are local registers and are not preserved for the caller. All other floating point registers are preserved. Note that, when compiling UEFI programs, a special switch will likely need to be specified to guarantee that the compiler does not use f32-f127, which are not normally preserved in the regular calling convention for Itanium. A procedure using one of the preserved floating point registers must save and restore the caller's original contents without generating a NaT consumption fault.

Floating point arguments are passed in f8-f15 registers when possible. Parameters beyond the registers appear in memory, as explained in Section 8.5 of the Itanium Software Conventions and Runtime Architecture Guide. Within the called function, these are local registers and are not preserved for the caller. Registers f6-f7 are local registers and are not preserved for the caller. All other floating point registers are preserved. Note that, when compiling UEFI programs, a special switch will likely need to be specified to guarantee that the compiler does not use f32-f127, which are not normally preserved in the regular calling convention for Itanium. A procedure using one of the preserved floating point registers must save and restore the caller's original contents without generating a NaT consumption fault.

The floating point status register must be preserved across calls to a target function. Flags fields in SF1,2,3 are not preserved for the caller. Flags fields in SF0 upon return will reflect the value passed in, and with bits set to 1 corresponding to any IEEE exceptions detected on non-speculative floating-point operations executed as part of the callee.

Floating-point operations executed by the callee may require software emulation. The caller must be prepared to handle FP Software Assist (FPSWA) interruptions. Callees should not raise IEEE traps by changing FPSR.traps bits to 0 and then executing floating-point operations that raise such traps.

2.3.4 x64 Platforms

All functions are called with the C language calling convention. See Section 2.3.4.2 for more detail.

During boot services time the processor is in the following execution mode:

- Uniprocessor, as described in chapter 8.4 of:
  - See “Links to UEFI-Related Documents” (http://uefi.org/uefi) under the heading “Intel Processor Manuals”.

- Long mode, in 64-bit mode

- Paging mode is enabled and any memory space defined by the UEFI memory map is identity mapped (virtual address equals physical address), although the attributes of certain regions may not have all read, write, and execute attributes or be unmarked for purposes of platform protection. The mappings to other regions, such as those for unaccepted memory, are undefined and may vary from implementation to implementation.
• Selectors are set to be flat and are otherwise not used.
• Interrupts are enabled—though no interrupt services are supported other than the UEFI boot services timer functions (All loaded device drivers are serviced synchronously by “polling.”)
• Direction flag in EFLAGS is clear
• Other general purpose flag registers are undefined
• 128 KiB, or more, of available stack space
• The stack must be 16-byte aligned. Stack may be marked as non-executable in identity mapped page tables.
• Floating-point control word must be initialized to 0x037F (all exceptions masked, double-extended-precision, round-to-nearest)
• Multimedia-extensions control word (if supported) must be initialized to 0x1F80 (all exceptions masked, round-to-nearest, flush to zero for masked underflow).
• CR0.EM must be zero
• CR0.TS must be zero

For an operating system to use any UEFI runtime services, it must:

• Preserve all memory in the memory map marked as runtime code and runtime data
• Call the runtime service functions, with the following conditions:
  • In long mode, in 64-bit mode
  • Paging enabled
  • All selectors set to be flat with virtual = physical address. If the UEFI OS loader or OS used SetVirtualAddressMap() to relocate the runtime services in a virtual address space, then this condition does not have to be met. See description of SetVirtualAddressMap() for details of memory map after this function has been called.
• Direction flag in EFLAGS clear
• 4 KiB, or more, of available stack space
• The stack must be 16-byte aligned
• Floating-point control word must be initialized to 0x037F (all exceptions masked, double-extended-precision, round-to-nearest)
• Multimedia-extensions control word (if supported) must be initialized to 0x1F80 (all exceptions masked, round-to-nearest, flush to zero for masked underflow)
• CR0.EM must be zero
• CR0.TS must be zero
• Interrupts may be disabled or enabled at the discretion of the caller.
• ACPI Tables loaded at boot time can be contained in memory of type EfiACPIReclaimMemory (recommended) or EfiACPIMemoryNVS. ACPI FACS must be contained in memory of type EfiACPIMemoryNVS.
• The system firmware must not request a virtual mapping for any memory descriptor of type EfiACPIReclaimMemory or EfiACPIMemoryNVS.
• EFI memory descriptors of type EfiACPIReclaimMemory and EfiACPIMemoryNVS must be aligned on a 4 KiB boundary and must be a multiple of 4 KiB in size.

• Any UEFI memory descriptor that requests a virtual mapping via the EFI_MEMORY_DESCRIPTOR having the EFI_MEMORY_RUNTIME bit set must be aligned on a 4 KiB boundary and must be a multiple of 4 KiB in size.

• An ACPI Memory Op-region must inherit cacheability attributes from the UEFI memory map. If the system memory map does not contain cacheability attributes, the ACPI Memory Op-region must inherit its cacheability attributes from the ACPI name space. If no cacheability attributes exist in the system memory map or the ACPI name space, then the region must be assumed to be non-cacheable.

• ACPI tables loaded at runtime must be contained in memory of type EfiACPIMemoryNVS. The cacheability attributes for ACPI tables loaded at runtime should be defined in the UEFI memory map. If no information about the table location exists in the UEFI memory map, cacheability attributes may be obtained from ACPI memory descriptors. If no information about the table location exists in the UEFI memory map or ACPI memory descriptors, the table is assumed to be non-cached.

• In general, UEFI Configuration Tables loaded at boot time (e.g., SMBIOS table) can be contained in memory of type EfiRuntimeServicesData (recommended), EfiBootServicesData, EfiACPIReclaimMemory, or EfiACPIMemoryNVS. Tables loaded at runtime must be contained in memory of type EfiRuntimeServicesData (recommended) or EfiACPIMemoryNVS.

Note: Previous EFI specifications allowed ACPI tables loaded at runtime to be in the EfiReservedMemoryType and there was no guidance provided for other EFI Configuration Tables. EfiReservedMemoryType is not intended to be used by firmware. Also, only OSes conforming to the UEFI Specification are guaranteed to handle SMBIOS table in memory of type EfiBootServicesData.

2.3.4.1 Handoff State

Rcx – EFI_HANDLE
Rdx – EFI_SYSTEM_TABLE *
RSP - <return address>

2.3.4.2 Detailed Calling Conventions

The caller passes the first four integer arguments in registers. The integer values are passed from left to right in Rcx, Rdx, R8, and R9 registers. The caller passes arguments five and above onto the stack. All arguments must be right-justified in the register in which they are passed. This ensures the callee can process only the bits in the register that are required.

The caller passes arrays and strings via a pointer to memory allocated by the caller. The caller passes structures and unions of size 8, 16, 32, or 64 bits as if they were integers of the same size. The caller is not allowed to pass structures and unions of other than these sizes and must pass these unions and structures via a pointer.

The callee must dump the register parameters into their shadow space if required. The most common requirement is to take the address of an argument.
If the parameters are passed through varargs then essentially the typical parameter passing applies, including spilling the fifth and subsequent arguments onto the stack. The callee must dump the arguments that have their address taken.

Return values that fix into 64-bits are returned in the Rax register. If the return value does not fit within 64-bits, then the caller must allocate and pass a pointer for the return value as the first argument, Rcx. Subsequent arguments are then shifted one argument to the right, so for example argument one would be passed in Rdx. User-defined types to be returned must be 1,2,4,8,16,32, or 64 bits in length.

The registers Rax, Rcx Rdx R8, R9, R10, R11, and XMM0-XMM5 are volatile and are, therefore, destroyed on function calls.

The registers RBX, RBP, RDI, RSI, R12, R13, R14, R15, and XMM6-XMM15 are considered nonvolatile and must be saved and restored by a function that uses them.

Function pointers are pointers to the label of the respective function and don’t require special treatment.

A caller must always call with the stack 16-byte aligned.

For MMX, XMM and floating-point values, return values that can fit into 64-bits are returned through RAX (including MMX types). However, XMM 128-bit types, floats, and doubles are returned in XMM0. The floating point status register is not saved by the target function. Floating-point and double-precision arguments are passed in XMM0 - XMM3 (up to 4) with the integer slot (RCX, RDX, R8, and R9) that would normally be used for that cardinal slot being ignored (see example) and vice versa. XMM types are never passed by immediate value but rather a pointer will be passed to memory allocated by the caller. MMX types will be passed as if they were integers of the same size. Callees must not unmask exceptions without providing correct exception handlers.

In addition, unless otherwise specified by the function definition, all other CPU registers (including MMX and XMM) are preserved.

2.3.4.3 Enabling Paging or Alternate Translations in an Application

Boot Services define an execution environment where paging is not enabled (supported 32-bit) or where translations are enabled but mapped virtual equal physical (x64) and this section will describe how to write an application with alternate translations or with paging enabled. Some Operating Systems require the OS Loader to be able to enable OS required translations at Boot Services time.

If a UEFI application uses its own page tables, GDT or IDT, the application must ensure that the firmware executes with each supplanted data structure. There are two ways that firmware conforming to this specification can execute when the application has paging enabled.

- Explicit firmware call
- Firmware preemption of application via timer event

An application with translations enabled can restore firmware required mapping before each UEFI call. However the possibility of preemption may require the translation enabled application to disable interrupts while alternate translations are enabled. It’s legal for the translation enabled application to enable interrupts if the application catches the interrupt and restores the EFI firmware environment prior to calling the UEFI interrupt ISR. After the UEFI ISR context is executed it will return to the translation enabled application context and restore any mappings required by the application.
2.3.5 AArch32 Platforms

All functions are called with the C language calling convention specified in Section 2.3.5.3. In addition, the invoking OSs can assume that unaligned access support is enabled if it is present in the processor.

During boot services time the processor is in the following execution mode:

- Unaligned access should be enabled if supported; Alignment faults are enabled otherwise.
- Uniprocessor.
- A privileged mode.
- The MMU is enabled (CP15 c1 System Control Register (SCTRL) SCTRL.M=1) and any RAM defined by the UEFI memory map is identity mapped (virtual address equals physical address). The mappings to other regions are undefined and may vary from implementation to implementation.
- The core will be configured as follows (common across all processor architecture revisions):
  - MMU enabled
  - Instruction and Data caches enabled
  - Access flag disabled
  - Translation remap disabled
  - Little endian mode
  - Domain access control mechanism (if supported) will be configured to check access permission bits in the page descriptor
  - Fast Context Switch Extension (FCSE) must be disabled

This will be achieved by:

- Configuring the CP15 c1 System Control Register (SCTRL) as follows: I=1, C=1, B=0, TRE=0, AFE=0, M=1
- Configuring the CP15 c3 Domain Access Control Register (DACR) to 0x33333333.
- Configuring the CP15 c1 System Control Register (SCTRL), A=1 on ARMv4 and ARMv5, A=0, U=1 on ARMv6 and ARMv7.

The state of other system control register bits is not dictated by this specification.

- Implementations of boot services will enable architecturally manageable caches and TLBs i.e., those that can be managed directly using CP15 operations using mechanisms and procedures defined in the ARM Architecture Reference Manual. They should not enable caches requiring platform information to manage or invoke non-architectural cache/TLB lockdown mechanisms
- MMU configuration–Implementations must use only 4k pages and a single translation base register. On devices supporting multiple translation base registers, TTBR0 must be used solely. The binding does not mandate whether page tables are cached or un-cached.

- On processors implementing the ARMv4 through ARMv6K architecture definitions, the core is additionally configured to disable extended page tables support, if present. This will be achieved by configuring the CP15 c1 System Control Register (SCTRL) as follows: XP=0
On processors implementing the ARMv7 and later architecture definitions, the core will be configured to enable the extended page table format and disable the TEX remap mechanism. This will be achieved by configuring the CP15 c1 System Control Register (SCTLR) as follows: XP=1, TRE=0

- Interrupts are enabled—though no interrupt services are supported other than the UEFI boot services timer functions (All loaded device drivers are serviced synchronously by “polling.”)
- 128 KiB or more of available stack space

For an operating system to use any runtime services, it must:

- Preserve all memory in the memory map marked as runtime code and runtime data
- Call the runtime service functions, with the following conditions:
  - In a privileged mode.
  - The system address regions described by all the entries in the EFI memory map that have the EFI_MEMORY_RUNTIME bit set must be identity mapped as they were for the EFI boot environment. If the OS Loader or OS used SetVirtualAddressMap() to relocate the runtime services in a virtual address space, then this condition does not have to be met. See description of SetVirtualAddressMap() for details of memory map after this function has been called.
  - The processor must be in a mode in which it has access to the system address regions specified in the EFI memory map with the EFI_MEMORY_RUNTIME bit set.
  - 4 KiB, or more, of available stack space
  - Interrupts may be disabled or enabled at the discretion of the caller

An application written to this specification may alter the processor execution mode, but the invoking OS must ensure firmware boot services and runtime services are executed with the prescribed execution environment.

If ACPI is supported:

- ACPI Tables loaded at boot time can be contained in memory of type EfiACPIReclaimMemory (recommended) or EfiACPIMemoryNVS. ACPI FACS must be contained in memory of type EfiACPIMemoryNVS
- The system firmware must not request a virtual mapping for any memory descriptor of type EfiACPIReclaimMemory or EfiACPIMemoryNVS.
- EFI memory descriptors of type EfiACPIReclaimMemory and EfiACPIMemoryNVS must be aligned on a 4 KiB boundary and must be a multiple of 4 KiB in size.
- Any UEFI memory descriptor that requests a virtual mapping via the EFI_MEMORY_DESCRIPTOR having the EFI_MEMORY_RUNTIME bit set must be aligned on a 4 KiB boundary and must be a multiple of 4 KiB in size.
- An ACPI Memory Op-region must inherit cacheability attributes from the UEFI memory map. If the system memory map does not contain cacheability attributes, the ACPI Memory Op-region must inherit its cacheability attributes from the ACPI name space. If no cacheability attributes exist in the system memory map or the ACPI name space, then the region must be assumed to be non-cacheable.
• ACPI tables loaded at runtime must be contained in memory of type EfiACPIMemoryNVS. The cacheability attributes for ACPI tables loaded at runtime should be defined in the UEFI memory map. If no information about the table location exists in the UEFI memory map, cacheability attributes may be obtained from ACPI memory descriptors. If no information about the table location exists in the UEFI memory map or ACPI memory descriptors, the table is assumed to be non-cached.

• In general, UEFI Configuration Tables loaded at boot time (e.g., SMBIOS table) can be contained in memory of type EfiRuntimeServicesData (recommended), EfiBootServicesData, EfiACPIReclaimMemory or EfiACPIMemoryNVS. Tables loaded at runtime must be contained in memory of type EfiRuntimeServicesData (recommended) or EfiACPIMemoryNVS.

Note: Previous EFI specifications allowed ACPI tables loaded at runtime to be in the EfiReservedMemoryType and there was no guidance provided for other EFI Configuration Tables. EfiReservedMemoryType is not intended to be used by firmware. Also, only OSes conforming to the UEFI Specification are guaranteed to handle SMBIOS table in memory of type EfiBootServicesData.

2.3.5.1 Handoff State

R0 – EFI_HANDLE
R1 – EFI_SYSTEM_TABLE *
R14 – Return Address

2.3.5.2 Enabling Paging or Alternate Translations in an Application

Boot Services define a specific execution environment. This section will describe how to write an application that creates an alternative execution environment. Some Operating Systems require the OS Loader to be able to enable OS required translations at Boot Services time, and make other changes to the UEFI defined execution environment.

If a UEFI application uses its own page tables, or other processor state, the application must ensure that the firmware executes with each supplanted functionality. There are two ways that firmware conforming to this specification can execute in this alternate execution environment:

• Explicit firmware call
• Firmware preemption of application via timer event

An application with an alternate execution environment can restore the firmware environment before each UEFI call. However the possibility of preemption may require the alternate execution-enabled application to disable interrupts while the alternate execution environment is active. It's legal for the alternate execution environment enabled application to enable interrupts if the application catches the interrupt and restores the EFI firmware environment prior to calling the UEFI interrupt ISR. After the UEFI ISR context is executed it will return to the alternate execution environment enabled application context.

An alternate execution environment created by a UEFI application must not change the semantics or behavior of the MMU configuration created by the UEFI firmware prior to invoking ExitBootServices(), including the bit layout of the page table entries.
After an OS loader calls `ExitBootServices()` it should immediately configure the exception vector to point to appropriate code.

### 2.3.5.3 Detailed Calling Convention

The base calling convention for the ARM binding is defined here:

*Procedure Call Standard for the ARM Architecture V2.06 (or later)*

See “Links to UEFI-Related Documents” ([http://uefi.org/uefi](http://uefi.org/uefi)) under the heading “Arm Architecture Base Calling Convention”.

This binding further constrains the calling convention in these ways:

- Calls to UEFI defined interfaces must be done assuming that the target code requires the ARM instruction set state. Images are free to use other instruction set states except when invoking UEFI interfaces.
- Floating point, SIMD, vector operations and other instruction set extensions must not be used.
- Only little endian operation is supported.
- The stack will maintain 8 byte alignment as described in the AAPCS for public interfaces.
- Use of coprocessor registers for passing call arguments must not be used.
- Structures (or other types larger than 64-bits) must be passed by reference and not by value.
- The EFI ARM platform binding defines register r9 as an additional callee-saved variable register.

### 2.3.6 AArch64 Platforms

AArch64 UEFI will only execute 64-bit ARM code, as the ARMv8 architecture does not allow for the mixing of 32-bit and 64-bit code at the same privilege level.

All functions are called with the C language calling convention specified in Detailed calling Convention section below. During boot services only a single processor is used for execution. All secondary processors must be either powered off or held in a quiescent state.

The primary processor is in the following execution mode:

- Unaligned access must be enabled.
- Use the highest 64 bit non secure privilege level available; Non-secure EL2 (Hyp) or Non-secure EL1(Kernel).
- The MMU is enabled and any RAM defined by the UEFI memory map is identity mapped (virtual address equals physical address). The mappings to other regions are undefined and may vary from implementation to implementation.
- The core will be configured as follows:
  - MMU enabled
  - Instruction and Data caches enabled
  - Little endian mode
  - Stack Alignment Enforced
  - NOT Top Byte Ignored
  - Valid Physical Address Space
• 4K Translation Granule

This will be achieved by:

1. Configuring the System Control Register SCTLR_EL2 or SCTLR_EL1:
   • EE=0, I=1, SA=1, C=1, A=0, M=1
2. Configuring the appropriate Translation Control Register:
   • TCR_EL2
     • TBI=0
     • PS must contain the valid Physical Address Space Size.
     • TG0=0
   • TCR_EL1
     • TBI0=0
     • IPS must contain the valid Intermediate Physical Address Space Size.
     • TG0=0

Note: The state of other system control register bits is not dictated by this specification.

• All floating point traps and exceptions will be disabled at the relevant exception levels (FPCR=0, CPACR_EL1.FPEN=11, CPTR_EL2.TFP=0). This implies that the FP unit will be enabled by default.
• Implementations of boot services will enable architecturally manageable caches and TLBs i.e., those that can be managed directly using implementation independent registers using mechanisms and procedures defined in the ARM Architecture Reference Manual. They should not enable caches requiring platform information to manage or invoke non-architectural cache/TLB lockdown mechanisms.
• MMU configuration: Implementations must use only 4k pages and a single translation base register. On devices supporting multiple translation base registers, TTBR0 must be used solely. The binding does not mandate whether page tables are cached or un-cached.
• Interrupts are enabled, though no interrupt services are supported other than the UEFI boot services timer functions (All loaded device drivers are serviced synchronously by “polling”). All UEFI interrupts must be routed to the IRQ vector only.
• The architecture generic timer must be initialized and enabled. The Counter Frequency register (CNTFRQ) must be programmed with the timer frequency. Timer access must be provided to non-secure EL1 and EL0 by setting bits EL1PCTEN and EL1PCEN in register CNTHCTL_EL2.
• The system firmware is not expected to initialize EL2 registers that do not have an architectural reset value, except in cases where firmware itself is running at EL2 and needs to do so.
• 128 KiB or more of available stack space
• The ARM architecture allows mapping pages at a variety of granularities, including 4KiB and 64KiB. If a 64KiB physical page contains any 4KiB page with any of the following types listed below, then all 4KiB pages in the 64KiB page must use identical ARM Memory Page Attributes (as described in Table 2-5):
  — EfiRuntimeServicesCode
  — EfiRuntimeServicesData
Mixed attribute mappings within a larger page are not allowed.

**Note:** This constraint allows a 64K paged based Operating System to safely map runtime services memory.

For an operating system to use any runtime services, Runtime services must:
- Support calls from either the EL1 or the EL2 exception levels.
- Once called, simultaneous or nested calls from EL1 and EL2 are not permitted.

**Note:** Sequential, non-overlapping, calls from EL1 and EL2 are permitted.

Runtime services are permitted to make synchronous SMC and HVC calls into higher exception levels.

**Note:** These rules allow Boot Services to start at EL2, and Runtime services to be assigned to an EL1 Operating System. In this case a call to `SetVirtualAddressMap()` is expected to provided an EL1 appropriate set of mappings.

For an operating system to use any runtime services, it must:
- Enable unaligned access support.
- Preserve all memory in the memory map marked as runtime code and runtime data
- Call the runtime service functions, with the following conditions:
  - From either EL1 or EL2 exception levels.
  - Consistently call runtime services from the same exception level. Sharing of runtime services between different exception levels is not permitted.
  - Runtime services must only be assigned to a single operating system or hypervisor. They must not be shared between multiple guest operating systems.
  - The system address regions described by all the entries in the EFI memory map that have the `EFI_MEMORY_RUNTIME` bit set must be identity mapped as they were for the EFI boot environment. If the OS Loader or OS used `SetVirtualAddressMap()` to relocate the runtime services in a virtual address space, then this condition does not have to be met. See description of `SetVirtualAddressMap()` for details of memory map after this function has been called.
  - The processor must be in a mode in which it has access to the system address regions specified in the EFI memory map with the `EFI_MEMORY_RUNTIME` bit set.
  - 8 KiB, or more, of available stack space.
  - The stack must be 16-byte aligned (128-bit).
  - Interrupts may be disabled or enabled at the discretion of the caller.

An application written to this specification may alter the processor execution mode, but the invoking OS must ensure firmware boot services and runtime services are executed with the prescribed execution environment.

If ACPI is supported:

- ACPI Tables loaded at boot time can be contained in memory of type
  `EfiACPIReclaimMemory` (recommended) or `EfiACPIMemoryNVS`.
- ACPI FACS must be contained in memory of type `EfiACPIMemoryNVS`. The system firmware
  must not request a virtual mapping for any memory descriptor of type
  `EfiACPIReclaimMemory` or `EfiACPIMemoryNVS`.
- EFI memory descriptors of type `EfiACPIReclaimMemory` and `EfiACPIMemoryNVS` must be
  aligned on a 4 KiB boundary and must be a multiple of 4 KiB in size.
- Any UEFI memory descriptor that requests a virtual mapping via the
  `EFI_MEMORY_DESCRIPTOR` having the `EFI_MEMORY_RUNTIME` bit set must be aligned on a 4
  KiB boundary and must be a multiple of 4 KiB in size.
- An ACPI Memory Op-region must inherit cacheability attributes from the UEFI memory map. If
  the system memory map does not contain cacheability attributes, the ACPI Memory Op-region
  must inherit its cacheability attributes from the ACPI name space. If no cacheability attributes
  exist in the system memory map or the ACPI name space, then the region must be assumed to
  be non-cacheable.
- ACPI tables loaded at runtime must be contained in memory of type `EfiACPIMemoryNVS`. The
  cacheability attributes for ACPI tables loaded at runtime should be defined in the UEFI memory
  map. If no information about the table location exists in the UEFI memory map, cacheability
  attributes may be obtained from ACPI memory descriptors. If no information about the table
  location exists in the UEFI memory map or ACPI memory descriptors, the table is assumed to
  be non-cached.
- In general, UEFI Configuration Tables loaded at boot time (e.g., SMBIOS table) can be contained
  in memory of type `EfiRuntimeServicesData` (recommended), `EfiBootServicesData`,
  `EfiACPIReclaimMemory` or `EfiACPIMemoryNVS`. Tables loaded at runtime must be
  contained in memory of type `EfiRuntimeServicesData` (recommended) or
  `EfiACPIMemoryNVS`.

**Note:** Previous EFI specifications allowed ACPI tables loaded at runtime to be in the
`EfiReservedMemoryType` and there was no guidance provided for other EFI Configuration
Tables. `EfiReservedMemoryType` is not intended to be used by firmware. UEFI 2.0 clarified the
situation moving forward. Also, only OSes conforming to UEFI Specification are guaranteed to
handle SMBIOS table in memory of type `EfiBootServiceData`. 
2.3.6.1 Memory types

Table 2-5 Map: EFI Cacheability Attributes to AArch64 Memory Types

<table>
<thead>
<tr>
<th>EFI Memory Type</th>
<th>ARM Memory Type: MAIR attribute encoding</th>
<th>ARM Memory Type: Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>EFI_MEMORY_UC (Not cacheable)</td>
<td>0000 0000</td>
<td>Device-nGnRnE (Device non-Gathering, non-Reordering, no Early Write Acknowledgement)</td>
</tr>
<tr>
<td>EFI_MEMORY_WC (Write combine)</td>
<td>0100 0100</td>
<td>Normal Memory Outer non-cacheable Inner non-cacheable</td>
</tr>
<tr>
<td>EFI_MEMORY_WT (Write through)</td>
<td>1011 1011</td>
<td>Normal Memory Outer Write-through non-transient Inner Write-through non-transient</td>
</tr>
<tr>
<td>EFI_MEMORY_WB (Write back)</td>
<td>1111 1111</td>
<td>Normal Memory Outer Write-back non-transient Inner Write-back non-transient</td>
</tr>
<tr>
<td>EFI_MEMORY_UCE</td>
<td></td>
<td>Not used or defined</td>
</tr>
</tbody>
</table>

Table 2-6 Map: UEFI Permission Attributes to ARM Paging Attributes

<table>
<thead>
<tr>
<th>EFI Memory Type</th>
<th>ARM Paging Attributes</th>
</tr>
</thead>
<tbody>
<tr>
<td>EFI_MEMORY_XP</td>
<td>EL2 translation regime: XN Execute never</td>
</tr>
<tr>
<td></td>
<td>EL1/0 translation regime: UXN Unprivileged execute never</td>
</tr>
<tr>
<td></td>
<td>PXN Privileged execute never</td>
</tr>
<tr>
<td>EFI_MEMORY_RO</td>
<td>Read only access AP[2]=1</td>
</tr>
<tr>
<td>EFI_MEMORY_RP</td>
<td>Not used or defined</td>
</tr>
<tr>
<td>EFI_MEMORY_WP</td>
<td>Not used or defined</td>
</tr>
</tbody>
</table>

2.3.6.2 Handoff State

X0 – EFI_HANDLE
X1 – EFI_SYSTEM_TABLE *
X30 – Return Address

2.3.6.3 Enabling Paging or Alternate Translations in an Application

Boot Services define a specific execution environment. This section will describe how to write an application that creates an alternative execution environment. Some Operating Systems require the OS Loader to be able to enable OS required translations at Boot Services time, and make other changes to the UEFI defined execution environment.
If a UEFI application uses its own page tables, or other processor state, the application must ensure that the firmware executes with each supplanted functionality. There are two ways that firmware conforming to this specification can execute in this alternate execution environment:

- Explicit firmware call
- Firmware preemption of application via timer event

An application with an alternate execution environment can restore the firmware environment before each UEFI call. However the possibility of preemption may require the alternate execution-enabled application to disable interrupts while the alternate execution environment is active. It's legal for the alternate execution environment enabled application to enable interrupts if the application catches the interrupt and restores the EFI firmware environment prior to calling the UEFI interrupt ISR. After the UEFI ISR context is executed it will return to the alternate execution environment enabled application context.

An alternate execution environment created by a UEFI application must not change the semantics or behavior of the MMU configuration created by the UEFI firmware prior to invoking `ExitBootServices()`, including the bit layout of the page table entries.

After an OS loader calls `ExitBootServices()` it should immediately configure the exception vector to point to appropriate code.

### 2.3.6.4 Detailed Calling Convention

The base calling convention for the AArch64 binding is defined in the document *Procedure Call Standard for the ARM 64-bit Architecture Version A-0.06 (or later)*:

See “Links to UEFI-Related Documents” ([http://uefi.org/uefi](http://uefi.org/uefi)) under the heading “ARM 64-bit Base Calling Convention”

This binding further constrains the calling convention in these ways:

- The AArch64 execution state must not be modified by the callee.
- All code exits, normal and exceptional, must be from the A64 instruction set.
- Floating point and SIMD instructions may be used.
- Optional vector operations and other instruction set extensions may only be used:
  - After dynamically checking for their existence.
  - Saving and then later restoring any additional execution state context.
  - Additional feature enablement or control, such as power, must be explicitly managed.
- Only little endian operation is supported.
- The stack will maintain 16 byte alignment.
- Structures (or other types larger than 64-bits) must be passed by reference and not by value.
- The EFI AArch64 platform binding defines the platform register (r18) as “do not use”. Avoiding use of r18 in firmware makes the code compatible with both a fixed role for r18 defined by the OS platform ABI and the use of r18 by the OS and its applications as a temporary register.

### 2.3.7 RISC-V Platforms

All functions are called with the C language calling convention. See 2.3.7.3 for more detail.
On RISC-V platform, three privileged levels are currently introduced in RISC-V architecture. Beyond the User privilege, Supervisor privilege and Machine privileges cover all aspects of RISC-V system. The privileged instructions are also defined in each privilege level.

<table>
<thead>
<tr>
<th>Level</th>
<th>Encoding</th>
<th>Name</th>
<th>Abbreviation</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>User/Application</td>
<td>U</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>Supervisor</td>
<td>S</td>
</tr>
<tr>
<td>2</td>
<td>10</td>
<td>Reserved</td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>11</td>
<td>Machine</td>
<td>M</td>
</tr>
</tbody>
</table>

A RISC-V platform can contain one or more RISC-V cores and other components such as physical memory, fixed-function accelerators and I/O devices. The term RISC-V core refers to a component which contains an independent instruction fetch unit. A RISC-V core may have multiple RISC-V-compatible hardware threads, or hart. RISC-V UEFI firmware could be executed in either Machine mode or Supervisor mode during the entire POST, according to the hart capability and the platform design. However, RISC-V UEFI firmware has to switch the boot hart to Supervisor mode at either early or late POST if the platform is designed to boot a Supervisor mode OS or OS loader.

The machine mode has the highest privilege and this mode is the only mandatory privilege level for RISC-V platforms; all other privilege levels are optional depending on the platform requirements. Machine mode is the initial privilege mode entered at the power-on reset. This level is used in UEFI for low-level access to a hardware platform.

UEFI firmware implementation may provide the Supervisor Binary Interface (SBI) to allow the Supervisor mode execution environment to invoke privileged functions or access privileged hardware.

The processor is in the following execution mode during boot service:

- Total 32 general-purpose registers x1-x31. Register x0 is hardwired to 0. Each register has its ABI (Application Binary Interface) name. See 2.3.7.3 for more detail.
- The width of the native base integer depends on the RISC-V privileged mode implementation. XLEN is a general term which used to refer the width of base integer in bits.
  - For the Base Integer ISA in 32-bit width, XLEN = 32
  - For the Base Integer ISA in 64-bit width, XLEN = 64
  - For the Base Integer ISA in 128-bit width, XLEN = 128
- The width of processor registers could be determined by placing the immediate 4 in a register then shifting the register left by 31 bits at a time. If zero after one shift, then the machine is RV32. If zero after two shifts, then the machine is RV64, else RV128.
- Processor reset vector is platform specified. In UEFI, it is configured to the platform implementation-defined reset vector. The reset vector address is the first instruction which fetched by RISC-V processor when the power-on reset.
- The mcause value after reset have implementation-specific interpretation, value 0 should be returned on implementations that do not distinguish different reset conditions. Implementations that distinguish different reset conditions should only use 0 to indicate the most complete reset (e.g., hard reset). The causes of reset could be power-on reset, external hard reset, brownout detected, watchdog timer elapse, sleep-mode wakeup, etc., which machine-mode UEFI system firmware has to distinguish.
- The mstatus.xIE indicates the current processor interrupt activation in current privilege mode.
  - mstatus.MIE is set to one while mstatus.SIE and mstatus.UIE are set to zero during early UEFI POST stage.
- The machine mode interrupt is enabled during boot service in UEFI. Two kinds of interrupts are enabled, one is for timer interrupt and another is software interrupt.
- mie.MSIE = 1
- mie.MTIE = 1
- The memory is in physical addressing mode. Page is disabled in RISC-V machine mode during UEFI boot service.
- I/O access is through memory map I/O.
- Only support Machine level Control and Status Registers (CSRs) in UEFI.
- Machine ISA (misa) register contains the information regarding to the capabilities of CPU implementation. The misa.MXL field encodes the native base integer ISA width in machine mode. MXLEN (Machine XLEN) is given by setting of misa.MXL.
  - misa.MXL = 1, MXLEN is 32 bit
  - misa.MXL = 2, MXLEN is 64 bit
  - misa.MXL = 3, MXLEN is 128 bit
- RISC-V processor supports extensive customization and specialization instruction sets. RISC-V variations provide various purposes of processor implementations and the processor capability is reported in the extension bits in in misa register. UEFI drivers will need to know the capabilities of processor before executing the specified RISC-V extension instructions. The extensions fields encodes the presence of the standard extensions, with a single bit per letter of the alphabet. (Bit 0 encodes presence of extension “A”, Bit 1 encodes presence of extension “B” and so on. Currently the single letter extension mnemonics are as below,
  - A – Atomic extension
  - B – Tentatively reserved for Bit operations extension
  - C – Compressed extension
  - D – Double-Precision Floating-Point extension
  - E – Reduced Register Set Indicator RV32E (16 registers)
  - F – Single-Precision Floating-Point extension
  - G – Additional standard extensions present
  - H – Hypervisor extension
  - I – RV32I/64I/128I base ISA
  - J – Tentatively reserved for Dynamically Translated Languages extension
  - K – Reserved
  - L – Tentatively reserved for Decimal Floating-Point extension
  - M – Integer Multiplication and Division extension
  - N – User-level interrupts supported
  - O – Reserved
  - P – Tentatively reserved for Packed-SIMD extension
  - Q – Quad-Precision Floating-Point extension
  - S – Supervisor mode implemented
— T - Tentatively reserved for Transactional Memory extension
— U - User mode implemented
— V - Tentatively reserved for Vector extension
— W - Reserved
— X - Non-standard extension present
— Y - Reserved
— Z - Reserved
— Zifenci – Instruction-Fetch Fence
— Zicsr – Control and Status Register Access

· Machine Vendor ID Register
  — The mvendorid is a 32-bit read-only register encoding the manufacture of the part. Value of 0 indicates this field is not implemented or this is a non-commercial implementation.

· Machine Architecture ID Register
  — The marchid is an MXLEN-bit read-only register encoding the base microarchitecture of the hart. The combination of mvendorid and marchid should uniquely identify the type of hart microarchitecture that is implemented.

· Machine Implementation ID Register
  — This provides a unique encoding of the version of processor implementation.

An application written to this specification may alter the processor execution mode, but the UEFI image must ensure firmware boot services and runtime services are executed with the prescribed execution environment.

After an Operating System calls ExitBootServices(), firmware boot services are no longer available and it is illegal to call any boot service. After ExitBootServices, firmware runtime services are still available and may be called with paging enabled and virtual address pointers if SetVirtualAddressMap() has been called describing all virtual address ranges used by the firmware runtime service.

If ACPI is supported:
• ACPI Tables loaded at boot time can be contained in memory of type EfiACPIReclaimMemory (recommended) or EfiACPIMemoryNVS. ACPI FACS must be contained in memory of type EfiACPIMemoryNVS
  • The system firmware must not request a virtual mapping for any memory descriptor of type EfiACPIReclaimMemory or EfiACPIMemoryNVS.
  • EFI memory descriptors of type EfiACPIReclaimMemory and EfiACPIMemoryNVS must be aligned on a 4 KiB boundary and must be a multiple of 4 KiB in size.
  • Any UEFI memory descriptor that requests a virtual mapping via the EFI_MEMORY_DESCRIPTOR having the EFI_MEMORY_RUNTIME bit set must be aligned on a 4 KiB boundary and must be a multiple of 4 KiB in size.
  • An ACPI Memory Op-region must inherit cacheability attributes from the UEFI memory map. If the system memory map does not contain cacheability attributes, the ACPI Memory Op-region must inherit its cacheability attributes from the ACPI name space. If no cacheability attributes exist in the
system memory map or the ACPI name space, then the region must be assumed to be non-cacheable.

- ACPI tables loaded at runtime must be contained in memory of type EfiACPIMemoryNVS.

The cacheability attributes for ACPI tables loaded at runtime should be defined in the UEFI memory map. If no information about the table location exists in the UEFI memory map, cacheability attributes may be obtained from ACPI memory descriptors. If no information about the table location exists in the UEFI memory map or ACPI memory descriptors, the table is assumed to be non-cached.

- In general, UEFI Configuration Tables loaded at boot time (e.g., SMBIOS table) can be contained in memory of type EfiRuntimeServicesData (recommended), EfiBootServicesData, EfiACPIReclaimMemory or EfiACPIMemoryNVS. Tables loaded at runtime must be contained in memory of type EfiRuntimeServicesData (recommended) or EfiACPIMemoryNVS.

**Note:** Previous EFI specifications allowed ACPI tables loaded at runtime to be in the EfiReservedMemoryType and there was no guidance provided for other EFI Configuration Tables. EfiReservedMemoryType is not intended to be used by firmware. The UEFI Specification intends to clarify the situation moving forward. Also, only OSes conforming to the UEFI Specification are guaranteed to handle SMBIOS table in memory of type EfiBootServicesData.

### 2.3.7.1 Handoff State

When UEFI firmware hands off control to Supervisor mode OS, RISC-V boot hart must be operated in Supervisor mode, and the memory addressing must be operated in Bare mode which is no memory address translation or protection through the virtual page table entry.

In order to describe the heterogeneous RISC-V cores and harts for the next boot stage after POST, UEFI firmware must build up the information of core and hart hardware capabilities in the firmware data structure if the target bootable image requires this information. (e.g. if the platform supports SMBIOS structure, SMBIOS record type 44 record, see “Link to UEFI Specification-Related Document” on https://uefi.org/uefi under the heading “RISC-V Processor SMBIOS Specification”).

UEFI firmware must exposes RISC-V boot hart ID in the firmware data structure to OS:

- If platform supports SMBIOS, then SMBIOS type 44 record for the boot hart must have “Boot Hart” set to 1 in “RISC-V Processor-specific Data” structure.

- If platform supports Device Tree, the Device Tree must contains a unsigned integer (32bit) property "boot- hartid" under the /chosen node which would indicate the booting hart ID to the supervisor OS.

If the platform supports Device Tree structure to describe the system configurations, the Flattened Device Blob (DTB) must be installed in the EFI Configuration Table (See **Section 4.6** for details).
All UEFI images takes two parameters: the UEFI image handle and the pointer to EFI System Table. According to the RISC-V calling convention, EFI_HANDLE is passed through the a0 register and EFI_SYSTEM_TABLE is passed through the a1 register.

- x10 – EFI_HANDLE (ABI name: a0)
- x11 – EFI_SYSTEM_TABLE *(ABI name: a1)
- x1 – Return Address (ABI name: ra)

### 2.3.7.2 Data alignment

In the RV32I and RV64I, the datatypes must be aligned at its natural size when stored in memory. The following table describes the datatype and its alignment in RV32I and RV64I in UEFI.

**Table 2-7 RV32 datatype alignment**

<table>
<thead>
<tr>
<th>Datatype</th>
<th>Description</th>
<th>Alignment</th>
</tr>
</thead>
<tbody>
<tr>
<td>BOOLEAN</td>
<td>Logical Boolean</td>
<td>1</td>
</tr>
<tr>
<td>INTN</td>
<td>Signed value in native width.</td>
<td>4</td>
</tr>
<tr>
<td>UINTN</td>
<td>Unsigned value in native width.</td>
<td>4</td>
</tr>
<tr>
<td>INT8</td>
<td>1-byte signed value</td>
<td>1</td>
</tr>
<tr>
<td>UINT8</td>
<td>1-byte unsigned value</td>
<td>1</td>
</tr>
<tr>
<td>INT16</td>
<td>2-byte signed value</td>
<td>2</td>
</tr>
<tr>
<td>UINT16</td>
<td>2-byte unsigned value</td>
<td>2</td>
</tr>
<tr>
<td>INT32</td>
<td>4-byte signed value</td>
<td>4</td>
</tr>
<tr>
<td>UINT32</td>
<td>4-byte unsigned value</td>
<td>4</td>
</tr>
<tr>
<td>INT64</td>
<td>8-byte signed value</td>
<td>8</td>
</tr>
<tr>
<td>UINT64</td>
<td>8-byte unsigned value</td>
<td>8</td>
</tr>
<tr>
<td>CHAR8</td>
<td>1-byte character</td>
<td>1</td>
</tr>
<tr>
<td>CHAR16</td>
<td>2-byte character</td>
<td>2</td>
</tr>
<tr>
<td>VOID</td>
<td>Undeclared type</td>
<td>4</td>
</tr>
</tbody>
</table>

**Table 2-8 RV64 datatype alignment**

<table>
<thead>
<tr>
<th>Datatype</th>
<th>Description</th>
<th>Alignment</th>
</tr>
</thead>
<tbody>
<tr>
<td>BOOLEAN</td>
<td>Logical Boolean</td>
<td>1</td>
</tr>
<tr>
<td>INTN</td>
<td>Signed value in native width.</td>
<td>8</td>
</tr>
<tr>
<td>UINTN</td>
<td>Unsigned value in native width.</td>
<td>8</td>
</tr>
<tr>
<td>INT8</td>
<td>1-byte signed value</td>
<td>1</td>
</tr>
<tr>
<td>UINT8</td>
<td>1-byte unsigned value</td>
<td>1</td>
</tr>
<tr>
<td>INT16</td>
<td>2-byte signed value</td>
<td>2</td>
</tr>
<tr>
<td>UINT16</td>
<td>2-byte unsigned value</td>
<td>2</td>
</tr>
<tr>
<td>INT32</td>
<td>4-byte signed value</td>
<td>4</td>
</tr>
<tr>
<td>UINT32</td>
<td>4-byte unsigned value</td>
<td>4</td>
</tr>
<tr>
<td>INT64</td>
<td>8-byte signed value</td>
<td>8</td>
</tr>
<tr>
<td>UINT64</td>
<td>8-byte unsigned value</td>
<td>8</td>
</tr>
<tr>
<td>CHAR8</td>
<td>1-byte character</td>
<td>1</td>
</tr>
<tr>
<td>CHAR16</td>
<td>2-byte character</td>
<td>2</td>
</tr>
<tr>
<td>VOID</td>
<td>Undeclared type</td>
<td>8</td>
</tr>
</tbody>
</table>

### 2.3.7.3 Detailed Calling Convention

The RISC-V calling convention passes arguments in register when necessary. In RISC-V, total 32 general registers are declared, each register has its corresponding ABI name.
In the RISC-V calling convention, up to eight integer registers are used for passing argument, a0-a7. a0-a7 are the ABI names and the corresponding registers are x10-x17. Values are returned from functions in integer registers a0 and a1, those are register x10 and x11. In the standard RISC-V calling convention, the stack grows downward and the stack point is always kept 16-byte aligned. Five integer register t0-t6 are temporary registers that are volatile across calls and must be saved by the caller if later used. Twelve integer registers s0-s11 are preserved across calls and must be saved by the callee if used.

In view of the following statement:

"In the standard ABI, procedures should not modify the integer registers tp and gp, because signal handlers may rely upon their values"

mentioned in the RISC-V EFL psABI Specification, and the RISC-V calling convention that gp and tp registers are not assigned a specific owner to save and restore their values (see links below), UEFI firmware must neither trust the values of tp and gp nor make an assumption of owning the write access to these register in any circumstances. (Such as in EFI Boot service, EFI Runtime service, EFI Management Mode service and any UEFI firmware interfaces which may invoked by the EFI drivers, OS or external firmware payload.)

Preserve the values in gp or tp register if UEFI firmware needs to change them, and never touch them after ExitBootServices(). Whether and how to preserve gp and tp in the UEFI firmware environment is implementation-specific.


### 2.4 Protocols

The protocols that a device handle supports are discovered through the EFI_BOOT_SERVICES.HandleProtocol() Boot Service or the EFI_BOOT_SERVICES.OpenProtocol() Boot Service. Each protocol has a specification that includes the following:

- The protocol’s globally unique ID (GUID)
- The Protocol Interface structure
- The Protocol Services
Unless otherwise specified a protocol’s interface structure is not allocated from runtime memory and the protocol member functions should not be called at runtime. If not explicitly specified a protocol member function can be called at a TPL level of less than or equal to TPL_NOTIFY (see Section 7.1). Unless otherwise specified a protocol’s member function is not reentrant or MP safe.

Any status codes defined by the protocol member function definition are required to be implemented, Additional error codes may be returned, but they will not be tested by standard compliance tests, and any software that uses the procedure cannot depend on any of the extended error codes that an implementation may provide.

To determine if the handle supports any given protocol, the protocol’s GUID is passed to HandleProtocol() or OpenProtocol(). If the device supports the requested protocol, a pointer to the defined Protocol Interface structure is returned. The Protocol Interface structure links the caller to the protocol-specific services to use for this device.

Figure 2-4 shows the construction of a protocol. The UEFI driver contains functions specific to one or more protocol implementations, and registers them with the Boot Service EFI_BOOT_SERVICES.InstallProtocolInterface(). The firmware returns the Protocol Interface for the protocol that is then used to invoke the protocol specific services. The UEFI driver keeps private, device-specific context with protocol interfaces.

The following C code fragment illustrates the use of protocols:
// There is a global “EffectsDevice” structure. This
// structure contains information pertinent to the device.

// Connect to the ILLUSTRATION_PROTOCOL on the EffectsDevice,
// by calling HandleProtocol with the device’s EFI device handle
// and the ILLUSTRATION_PROTOCOL GUID.

EffectsDevice.Handle = DeviceHandle;
Status = HandleProtocol (  
    EffectsDevice.EFIHandle,  
    &IllustrationProtocolGuid,  
    &EffectsDevice.IllustrationProtocol  
);

// Use the EffectsDevice illustration protocol's “MakeEffects”
// service to make flashy and noisy effects.

Status = EffectsDevice.IllustrationProtocol->MakeEffects (  
    EffectsDevice.IllustrationProtocol,  
    TheFlashyAndNoisyEffect  
);

Table 2-10 lists the UEFI protocols defined by this specification.

<table>
<thead>
<tr>
<th>Protocol</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>EFI_LOADED_IMAGE_PROTOCOL</td>
<td>Provides information on the image.</td>
</tr>
<tr>
<td>EFI_LOADED_IMAGE_DEVICE_PATH_PROTOCOL</td>
<td>Specifies the device path that was used when a PE/COFF image was loaded through the EFI Boot Service LoadImage().</td>
</tr>
<tr>
<td>EFI_DEVICE_PATH_PROTOCOL</td>
<td>Provides the location of the device.</td>
</tr>
<tr>
<td>EFI_DRIVER_BINDING_PROTOCOL</td>
<td>Provides services to determine if an UEFI driver supports a given controller, and services to start and stop a given controller.</td>
</tr>
<tr>
<td>EFI_DRIVER_FAMILY_OVERRIDE_PROTOCOL</td>
<td>Provides the Driver Family Override mechanism for selecting the best driver for a given controller.</td>
</tr>
<tr>
<td>EFI_PLATFORM_DRIVER_OVERRIDE_PROTOCOL</td>
<td>Provide a platform specific override mechanism for the selection of the best driver for a given controller.</td>
</tr>
<tr>
<td>EFI_BUS_SPECIFIC_DRIVER_OVERRIDE_PROTOCOL</td>
<td>Provides a bus specific override mechanism for the selection of the best driver for a given controller.</td>
</tr>
<tr>
<td>EFI_DRIVER_DIAGNOSTICS2_PROTOCOL</td>
<td>Provides diagnostics services for the controllers that UEFI drivers are managing.</td>
</tr>
<tr>
<td>EFI_COMPONENT_NAME2_PROTOCOL</td>
<td>Provides human readable names for UEFI Drivers and the controllers that the drivers are managing.</td>
</tr>
<tr>
<td>EFI_SIMPLE_TEXT_INPUT_PROTOCOL</td>
<td>Protocol interfaces for devices that support simple console style text input.</td>
</tr>
<tr>
<td>Protocol</td>
<td>Description</td>
</tr>
<tr>
<td>---------------------------------------------------------------</td>
<td>-----------------------------------------------------------------------------</td>
</tr>
<tr>
<td>EFI_SIMPLE_TEXT_OUTPUT_PROTOCOL</td>
<td>Protocol interfaces for devices that support console style text displaying.</td>
</tr>
<tr>
<td>EFI_SIMPLE_POINTER_PROTOCOL</td>
<td>Protocol interfaces for devices such as mice and trackballs.</td>
</tr>
<tr>
<td>EFI_SERIAL_IO_PROTOCOL</td>
<td>Protocol interfaces for devices that support serial character transfer.</td>
</tr>
<tr>
<td>EFI_LOAD_FILE_PROTOCOL</td>
<td>Protocol interface for reading a file from an arbitrary device.</td>
</tr>
<tr>
<td>EFI_LOAD_FILE2_PROTOCOL</td>
<td>Protocol interface for reading a non-boot option file from an arbitrary device</td>
</tr>
<tr>
<td>EFI_SIMPLE_FILE_SYSTEM_PROTOCOL</td>
<td>Protocol interfaces for opening disk volume containing a UEFI file system.</td>
</tr>
<tr>
<td>EFI_FILE_PROTOCOL</td>
<td>Provides access to supported file systems.</td>
</tr>
<tr>
<td>EFI_DISK_IO_PROTOCOL</td>
<td>A protocol interface that layers onto any BLOCK_IO or BLOCK_IO_EX interface.</td>
</tr>
<tr>
<td>EFI_BLOCK_IO_PROTOCOL</td>
<td>Protocol interfaces for devices that support block I/O style accesses.</td>
</tr>
<tr>
<td>EFI_BLOCK_IO2_PROTOCOL</td>
<td>Protocol interfaces for devices that support block I/O style accesses. This interface is capable of non-blocking transactions.</td>
</tr>
<tr>
<td>EFI_UNICODE_COLLATION_PROTOCOL</td>
<td>Protocol interfaces for string comparison operations.</td>
</tr>
<tr>
<td>EFI_PCI_ROOT_BRIDGE_IO_PROTOCOL</td>
<td>Protocol interfaces to abstract memory, I/O, PCI configuration, and DMA accesses to a PCI root bridge controller.</td>
</tr>
<tr>
<td>EFI_PCI_IO_PROTOCOL</td>
<td>Protocol interfaces to abstract memory, I/O, PCI configuration, and DMA accesses to a PCI controller on a PCI bus.</td>
</tr>
<tr>
<td>EFI_USB_IO_PROTOCOL</td>
<td>Protocol interfaces to abstract access to a USB controller.</td>
</tr>
<tr>
<td>EFI_SIMPLE_NETWORK_PROTOCOL</td>
<td>Provides interface for devices that support packet based transfers.</td>
</tr>
<tr>
<td>EFI_PXE_BASE_CODE_PROTOCOL</td>
<td>Protocol interfaces for devices that support network booting.</td>
</tr>
<tr>
<td>EFI_BIS_PROTOCOL</td>
<td>Protocol interfaces to validate boot images before they are loaded and invoked.</td>
</tr>
<tr>
<td>EFI_DEBUG_SUPPORT_PROTOCOL</td>
<td>Protocol interfaces to save and restore processor context and hook processor exceptions.</td>
</tr>
<tr>
<td>EFI_DEBUGPORT_PROTOCOL</td>
<td>Protocol interface that abstracts a byte stream connection between a debug host and a debug target system.</td>
</tr>
<tr>
<td>EFI_DECOMPRESS_PROTOCOL</td>
<td>Protocol interfaces to decompress an image that was compressed using the EFI Compression Algorithm.</td>
</tr>
<tr>
<td>EFI_EBC_PROTOCOL</td>
<td>Protocols interfaces required to support an EFI Byte Code interpreter.</td>
</tr>
<tr>
<td>EFI_GRAPHICS_OUTPUT_PROTOCOL</td>
<td>Protocol interfaces for devices that support graphical output.</td>
</tr>
<tr>
<td>EFI_NVM_EXPRESS_PASS_THRU_PROTOCOL</td>
<td>Protocol interfaces that allow NVM Express commands to be issued to an NVM Express controller.</td>
</tr>
<tr>
<td>EFI_EXT_SCSI_PASS_THRU_PROTOCOL</td>
<td>Protocol interfaces for a SCSI channel that allows SCSI Request Packets to be sent to SCSI devices.</td>
</tr>
<tr>
<td>Protocol</td>
<td>Description</td>
</tr>
<tr>
<td>----------------------------------------------</td>
<td>---------------------------------------------------------------------------------------------------------------------------------------------</td>
</tr>
<tr>
<td>EFI_USB2_HC_PROTOCOL</td>
<td>Protocol interfaces to abstract access to a USB Host Controller.</td>
</tr>
<tr>
<td>EFI_AUTHENTICATION_INFO_PROTOCOL</td>
<td>Provides access for generic authentication information associated with specific device paths.</td>
</tr>
<tr>
<td>EFI_DEVICE_PATH_UTILITIES_PROTOCOL</td>
<td>Aids in creating and manipulating device paths.</td>
</tr>
<tr>
<td>EFI_DEVICE_PATH_TO_TEXT_PROTOCOL</td>
<td>Converts device nodes and paths to text.</td>
</tr>
<tr>
<td>EFI_DEVICE_PATH_FROM_TEXT_PROTOCOL</td>
<td>Converts text to device paths and device nodes.</td>
</tr>
<tr>
<td>EFI_EDID_DISCOVERED_PROTOCOL</td>
<td>Contains the EDID information retrieved from a video output device.</td>
</tr>
<tr>
<td>EFI_EDID_ACTIVE_PROTOCOL</td>
<td>Contains the EDID information for an active video output device.</td>
</tr>
<tr>
<td>EFI_EDID_OVERRIDE_PROTOCOL</td>
<td>Produced by the platform to allow the platform to provide EDID information to the producer of the Graphics Output protocol</td>
</tr>
<tr>
<td>EFI_ISCSI_INITIATOR_NAME_PROTOCOL</td>
<td>Sets and obtains the iSCSI Initiator Name.</td>
</tr>
<tr>
<td>EFI_TAPE_IO_PROTOCOL</td>
<td>Provides services to control and access a tape drive.</td>
</tr>
<tr>
<td>EFI_MANAGED_NETWORK_PROTOCOL</td>
<td>Used to locate communication devices that are supported by an MNP driver and create and destroy instances of the MNP child protocol driver that can use the underlying communications devices.</td>
</tr>
<tr>
<td>EFI_ARP_SERVICE_BINDING_PROTOCOL</td>
<td>Used to locate communications devices that are supported by an ARP driver and to create and destroy instances of the ARP child protocol driver.</td>
</tr>
<tr>
<td>EFI_ARP_PROTOCOL</td>
<td>Used to resolve local network protocol addresses into network hardware addresses.</td>
</tr>
<tr>
<td>EFI_DHCP4_SERVICE_BINDING_PROTOCOL</td>
<td>Used to locate communication devices that are supported by an EFI DHCPv4 Protocol driver and to create and destroy EFI DHCPv4 Protocol child driver instances that can use the underlying communications devices.</td>
</tr>
<tr>
<td>EFI_DHCP4_PROTOCOL</td>
<td>Used to collect configuration information for the EFI IPv4 Protocol drivers and to provide DHCPv4 server and PXE boot server discovery services.</td>
</tr>
<tr>
<td>EFI_TCP4_SERVICE_BINDING_PROTOCOL</td>
<td>Used to locate EFI TCPv4Protocol drivers to create and destroy child of the driver to communicate with other host using TCP protocol.</td>
</tr>
<tr>
<td>EFI_TCP4_PROTOCOL</td>
<td>Provides services to send and receive data stream.</td>
</tr>
<tr>
<td>EFI_IP4_SERVICE_BINDING_PROTOCOL</td>
<td>Used to locate communication devices that are supported by an EFI IPv4 Protocol Driver and to create and destroy instances of the EFI IPv4 Protocol child protocol driver that can use the underlying communication device.</td>
</tr>
<tr>
<td>EFI_IP4_PROTOCOL</td>
<td>Provides basic network IPv4 packet I/O services.</td>
</tr>
<tr>
<td>EFI_IP4_CONFIG_PROTOCOL</td>
<td>The EFI IPv4 Config Protocol driver performs platform- and policy-dependent configuration of the EFI IPv4 Protocol driver.</td>
</tr>
</tbody>
</table>
2.5 UEFI Driver Model

The UEFI Driver Model is intended to simplify the design and implementation of device drivers, and produce small executable image sizes. As a result, some complexity has been moved into bus drivers and in a larger part into common firmware services.

A device driver is required to produce a Driver Binding Protocol on the same image handle on which the driver was loaded. It then waits for the system firmware to connect the driver to a controller. When that occurs, the device driver is responsible for producing a protocol on the controller’s device handle that abstracts the I/O operations that the controller supports. A bus driver performs these exact same tasks. In addition, a bus driver is also responsible for discovering any child controllers on the bus, and creating a device handle for each child controller found.

One assumption is that the architecture of a system can be viewed as a set of one or more processors connected to one or more core chipsets. The core chipsets are responsible for producing one or more I/O buses. The UEFI Driver Model does not attempt to describe the processors or the core chipsets. Instead, the UEFI Driver Model describes the set of I/O buses produced by the core chipsets, and any children of these I/O buses. These children can either be devices or additional I/O buses. This can be viewed as a tree of buses and devices with the core chipsets at the root of that tree.

The leaf nodes in this tree structure are peripherals that perform some type of I/O. This could include keyboards, displays, disks, network, etc. The nonleaf nodes are the buses that move data between devices and buses, or between different bus types. Figure 2-5 shows a sample desktop system with four buses and six devices.

<table>
<thead>
<tr>
<th>Protocol</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>EFI_IP4_CONFIG2_PROTOCOL</td>
<td>The EFI IPv4 Configuration II Protocol driver performs platform- and policy-dependent configuration of the EFI IPv4 Protocol driver.</td>
</tr>
<tr>
<td>EFI_UPD4_SERVICE_BINDING_PROTOCOL</td>
<td>Used to locate communication devices that are supported by an EFI UDPv4 Protocol driver and to create and destroy instances of the EFI UDPv4 Protocol child protocol driver that can use the underlying communication device.</td>
</tr>
<tr>
<td>EFI_UPD4_PROTOCOL</td>
<td>Provides simple packet-oriented services to transmit and receive UDP packets.</td>
</tr>
<tr>
<td>EFI_MTFTP4_SERVICE_BINDING_PROTOCOL</td>
<td>Used to locate communication devices that are supported by an EFI MTFTPv4 Protocol driver and to create and destroy instances of the EFI MTFTPv4 Protocol child protocol driver that can use the underlying communication device.</td>
</tr>
<tr>
<td>EFI_MTFTP4_PROTOCOL</td>
<td>Provides basic services for client-side unicast or multicast TFTP operations.</td>
</tr>
<tr>
<td>EFI_HASH_PROTOCOL</td>
<td>Allows creating a hash of an arbitrary message digest using one or more hash algorithms.</td>
</tr>
<tr>
<td>EFI_HASH_SERVICE_BINDING_PROTOCOL</td>
<td>Used to locate hashing services support provided by a driver and create and destroy instances of the EFI Hash Protocol so that a multiple drivers can use the underlying hashing services.</td>
</tr>
<tr>
<td>EFI_SD_MMC_PASS_THRU_PROTOCOL</td>
<td>Protocol interface that allows SD/eMMC commands to be sent to an SD/eMMC controller.</td>
</tr>
</tbody>
</table>
Figure 2-5 Desktop System

Figure 2-6 is an example of a more complex server system. The idea is to make the UEFI Driver Model simple and extensible so more complex systems like the one below can be described and managed in the preboot environment. This system contains six buses and eight devices.

Figure 2-6 Server System

The combination of firmware services, bus drivers, and device drivers in any given platform is likely to be produced by a wide variety of vendors including OEMs, IBVs, and IHVs. These different components from different vendors are required to work together to produce a protocol for an I/O device than can be used to boot a UEFI compliant operating system. As a result, the UEFI Driver Model is described in great detail in order to increase the interoperability of these components.
This remainder of this section is a brief overview of the UEFI Driver Model. It describes the legacy option ROM issues that the UEFI Driver Model is designed to address, the entry point of a driver, host bus controllers, properties of device drivers, properties of bus drivers, and how the UEFI Driver Model can accommodate hot-plug events.

2.5.1 Legacy Option ROM Issues

Legacy option ROMs have a number of constraints and limitations that restrict innovation on the part of platform designers and adapter vendors. At the time of writing, both ISA and PCI adapters use legacy option ROMs. For the purposes of this discussion, only PCI option ROMs will be considered; legacy ISA option ROMs are not supported as part of the UEFI Specification.

The following is a list of the major constraints and limitations of legacy option ROMs. For each issue, the design considerations that went into the design of the UEFI Driver Model are also listed. Thus, the design of the UEFI Driver Model directly addresses the requirements for a solution to overcome the limitations implicit to PC-AT-style legacy option ROMs.

2.5.1.1 32-bit/16-Bit Real Mode Binaries

Legacy option ROMs typically contain 16-bit real mode code for an IA-32 processor. This means that the legacy option ROM on a PCI card cannot be used in platforms that do not support the execution of IA-32 real mode binaries. Also, 16-bit real mode only allows the driver to access directly the lower 1 MiB of system memory. It is possible for the driver to switch the processor into modes other than real mode in order to access resources above 1 MiB, but this requires a lot of additional code, and causes interoperability issues with other option ROMs and the system BIOS. Also, option ROMs that switch the processor into to alternate execution modes are not compatible with Itanium Processors.

UEFI Driver Model design considerations:

- Drivers need flat memory mode with full access to system components.
- Drivers need to be written in C so they are portable between processor architectures.
- Drivers may be compiled into a virtual machine executable, allowing a single binary driver to work on machines using different processor architectures.

2.5.1.2 Fixed Resources for Working with Option ROMs

Since legacy option ROMs can only directly address the lower 1 MiB of system memory, this means that the code from the legacy option ROM must exist below 1 MiB. In a PC-AT platform, memory from 0x00000-0x9FFFF is system memory. Memory from 0xA0000-0xBFFFF is VGA memory, and memory from 0xF0000-0xFFFFF is reserved for the system BIOS. Also, since system BIOS has become more complex over the years, many platforms also use 0xE0000-0xEFFFF for system BIOS. This leaves 128 KiB of memory from 0xC0000-0xDFFFF for legacy option ROMs. This limits how many legacy option ROMs can be run during BIOS POST.

Also, it is not easy for legacy option ROMs to allocate system memory. Their choices are to allocate memory from Extended BIOS Data Area (EBDA), allocate memory through a Post Memory Manager (PMM), or search for free memory based on a heuristic. Of these, only EBDA is standard, and the others are not used consistently between adapters, or between BIOS vendors, which adds complexity and the potential for conflicts.
UEFI Driver Model design considerations:

- Drivers need flat memory mode with full access to system components.
- Drivers need to be capable of being relocated so that they can be loaded anywhere in memory (PE/COFF Images)
- Drivers should allocate memory through the boot services. These are well-specified interfaces, and can be guaranteed to function as expected across a wide variety of platform implementations.

2.5.1.3 Matching Option ROMs to their Devices

It is not clear which controller may be managed by a particular legacy option ROM. Some legacy option ROMs search the entire system for controllers to manage. This can be a lengthy process depending on the size and complexity of the platform. Also, due to limitation in BIOS design, all the legacy option ROMs must be executed, and they must scan for all the peripheral devices before an operating system can be booted. This can also be a lengthy process, especially if SCSI buses must be scanned for SCSI devices. This means that legacy option ROMs are making policy decision about how the platform is being initialized, and which controllers are managed by which legacy option ROMs. This makes it very difficult for a system designer to predict how legacy option ROMs will interact with each other. This can also cause issues with on-board controllers, because a legacy option ROM may incorrectly choose to manage the on-board controller.

UEFI Driver Model design considerations:

- Driver to controller matching must be deterministic
- Give OEMs more control through Platform Driver Override Protocol and Driver Configuration Protocol
- It must be possible to start only the drivers and controllers required to boot an operating system.

2.5.1.4 Ties to PC-AT System Design

Legacy option ROMs assume a PC-AT-like system architecture. Many of them include code that directly touches hardware registers. This can make them incompatible on legacy-free and headless platforms. Legacy option ROMs may also contain setup programs that assume a PC-AT-like system architecture to interact with a keyboard or video display. This makes the setup application incompatible on legacy-free and headless platforms.

UEFI Driver Model design considerations:

- Drivers should use well-defined protocols to interact with system hardware, system input devices, and system output devices.

2.5.1.5 Ambiguities in Specification and Workarounds Born of Experience

Many legacy option ROMs and BIOS code contain workarounds because of incompatibilities between legacy option ROMs and system BIOS. These incompatibilities exist in part because there are no clear specifications on how to write a legacy option ROM or write a system BIOS.
Also, interrupt chaining and boot device selection is very complex in legacy option ROMs. It is not always clear which device will be the boot device for the OS.

**UEFI Driver Model** design considerations:

- Drivers and firmware are written to follow this specification. Since both components have a clearly defined specification, compliance tests can be developed to prove that drivers and system firmware are compliant. This should eliminate the need to build workarounds into either drivers or system firmware (other than those that might be required to address specific hardware issues).
- Give OEMs more control through Platform Driver Override Protocol and Driver Configuration Protocol and other OEM value-add components to manage the boot device selection process.

### 2.5.2 Driver Initialization

The file for a driver image must be loaded from some type of media. This could include ROM, FLASH, hard drives, floppy drives, CD-ROM, or even a network connection. Once a driver image has been found, it can be loaded into system memory with the boot service `EFI_BOOT_SERVICE.LoadImage()`.

`LoadImage()` loads a PE/COFF formatted image into system memory. A handle is created for the driver, and a Loaded Image Protocol instance is placed on that handle. A handle that contains a Loaded Image Protocol instance is called an *Image Handle*. At this point, the driver has not been started. It is just sitting in memory waiting to be started. **Figure 2-7** shows the state of an image handle for a driver after `LoadImage()` has been called.

![Image Handle](imagehandle.png)

**Figure 2-7 Image Handle**

After a driver has been loaded with the boot service `LoadImage()`, it must be started with the boot service `EFI_BOOT_SERVICE.StartImage()`. This is true of all types of UEFI Applications and UEFI Drivers that can be loaded and started on an UEFI-compliant system. The entry point for a driver that follows the UEFI **Driver Model** must follow some strict rules. First, it is not allowed to touch any hardware. Instead, the driver is only allowed to install protocol instances onto its own *Image Handle*. A driver that follows the UEFI **Driver Model** is *required* to install an instance of the Driver Binding Protocol onto its own *Image Handle*. It may optionally install the Driver Configuration Protocol, the Driver Diagnostics Protocol, or the Component Name Protocol. In addition, if a driver wishes to be unloadable it may optionally update the Loaded Image Protocol (see Section 9) to provide its own `Unload()` function. Finally, if a driver needs to perform any special operations when the boot service `EFI_BOOT_SERVICE.ExitBootServices()` is called, it may optionally create an event with a notification function that is triggered when the boot service `ExitBootServices()` is called. An *Image Handle* that contains a Driver Binding Protocol instance is known as a *Driver Image Handle*. **Figure 2-8**
shows a possible configuration for the Image Handle from Figure 2-7 after the boot service
StartImage() has been called.

![Image Handle Diagram]

**Figure 2-8 Driver Image Handle**

### 2.5.3 Host Bus Controllers

Drivers are not allowed to touch any hardware in the driver’s entry point. As a result, drivers will be
loaded and started, but they will all be waiting to be told to manage one or more controllers in the
system. A platform component, like the Boot Manager, is responsible for managing the connection of
drivers to controllers. However, before even the first connection can be made, there has to be some
initial collection of controllers for the drivers to manage. This initial collection of controllers is known as
the Host Bus Controllers. The I/O abstractions that the Host Bus Controllers provide are produced by
firmware components that are outside the scope of the UEFI Driver Model. The device handles for the
Host Bus Controllers and the I/O abstraction for each one must be produced by the core firmware on the
platform, or a driver that may not follow the UEFI Driver Model. See the PCI Root Bridge I/O Protocol
Specification for an example of an I/O abstraction for PCI buses.

A platform can be viewed as a set of processors and a set of core chipset components that may produce
one or more host buses. Figure 2-9 shows a platform with \( n \) processors (CPUs), and a set of core chipset
components that produce \( m \) host bridges.
Each host bridge is represented in UEFI as a device handle that contains a Device Path Protocol instance, and a protocol instance that abstracts the I/O operations that the host bus can perform. For example, a PCI Host Bus Controller supports one or more PCI Root Bridges that are abstracted by the PCI Root Bridge I/O Protocol. Figure 2-10 shows an example device handle for a PCI Root Bridge.

A PCI Bus Driver could connect to this PCI Root Bridge, and create child handles for each of the PCI devices in the system. PCI Device Drivers should then be connected to these child handles, and produce I/O abstractions that may be used to boot a UEFI compliant OS. The following section describes the different types of drivers that can be implemented within the UEFI Driver Model. The UEFI Driver Model is very flexible, so all the possible types of drivers will not be discussed here. Instead, the major types will be covered that can be used as a starting point for designing and implementing additional driver types.
2.5.4 Device Drivers

A device driver is not allowed to create any new device handles. Instead, it installs additional protocol interfaces on an existing device handle. The most common type of device driver will attach an I/O abstraction to a device handle that was created by a bus driver. This I/O abstraction may be used to boot a UEFI compliant OS. Some example I/O abstractions would include Simple Text Output, Simple Input, Block I/O, and Simple Network Protocol. Figure 2-11 shows a device handle before and after a device driver is connected to it. In this example, the device handle is a child of the XYZ Bus, so it contains an XYZ I/O Protocol for the I/O services that the XYZ bus supports. It also contains a Device Path Protocol that was placed there by the XYZ Bus Driver. The Device Path Protocol is not required for all device handles. It is only required for device handles that represent physical devices in the system. Handles for virtual devices will not contain a Device Path Protocol.

The device driver that connects to the device handle in Figure 2-11 must have installed a Driver Binding Protocol on its own image handle. The Driver Binding Protocol (see Section 11.1) contains three functions called Supported(), Start(), and Stop(). The Supported() function tests to see if the driver supports a given controller. In this example, the driver will check to see if the device handle supports the Device Path Protocol and the XYZ I/O Protocol. If a driver’s Supported() function passes, then the driver can be connected to the controller by calling the driver’s Start() function. The Start() function is what actually adds the additional I/O protocols to a device handle. In this example, the Block I/O Protocol is being installed. To provide symmetry, the Driver Binding Protocol also has a Stop().
function that forces the driver to stop managing a device handle. This will cause the device driver to uninstall any protocol interfaces that were installed in Start().

The Supported(), Start(), and Stop() functions of the EFI Driver Binding Protocol are required to make use of the boot service EFI_BOOT_SERVICES.OpenProtocol() to get a protocol interface and the boot service EFI_BOOT_SERVICES.CloseProtocol() to release a protocol interface. OpenProtocol() and CloseProtocol() update the handle database maintained by the system firmware to track which drivers are consuming protocol interfaces. The information in the handle database can be used to retrieve information about both drivers and controllers. The new boot service EFI_BOOT_SERVICES.OpenProtocolInformation() can be used to get the list of components that are currently consuming a specific protocol interface.

2.5.5 Bus Drivers

Bus drivers and device drivers are virtually identical from the UEFI Driver Model’s point of view. The only difference is that a bus driver creates new device handles for the child controllers that the bus driver discovers on its bus. As a result, bus drivers are slightly more complex than device drivers, but this in turn simplifies the design and implementation of device drivers. There are two major types of bus drivers. The first creates handles for all child controllers on the first call to Start(). The other type allows the handles for the child controllers to be created across multiple calls to Start(). This second type of bus driver is very useful in supporting a rapid boot capability. It allows a few child handles or even one child handle to be created. On buses that take a long time to enumerate all of their children (e.g. SCSI), this can lead to a very large timesaving in booting a platform. Figure 2-12 shows the tree structure of a bus controller before and after Start() is called. The dashed line coming into the bus controller node represents a link to the bus controller’s parent controller. If the bus controller is a Host Bus Controller, then it will not have a parent controller. Nodes A, B, C, D, and E represent the child controllers of the bus controller.

![Figure 2-12 Connecting Bus Drivers](OM13153)
A bus driver that supports creating one child on each call to \texttt{Start()} might choose to create child C first, and then child E, and then the remaining children A, B, and D. The \texttt{Supported()}, \texttt{Start()}, and \texttt{Stop()} functions of the Driver Binding Protocol are flexible enough to allow this type of behavior.

A bus driver must install protocol interfaces onto every child handle that is created. At a minimum, it must install a protocol interface that provides an I/O abstraction of the bus’s services to the child controllers. If the bus driver creates a child handle that represents a physical device, then the bus driver must also install a Device Path Protocol instance onto the child handle. A bus driver may optionally install a Bus Specific Driver Override Protocol onto each child handle. This protocol is used when drivers are connected to the child controllers. The boot service \texttt{EFI_BOOT_SERVICES.ConnectController()} uses architecturally defined precedence rules to choose the best set of drivers for a given controller. The Bus Specific Driver Override Protocol has higher precedence than a general driver search algorithm, and lower precedence than platform overrides. An example of a bus specific driver selection occurs with PCI. A PCI Bus Driver gives a driver stored in a PCI controller’s option ROM a higher precedence than drivers stored elsewhere in the platform. \texttt{Figure 2-13} shows an example child device handle that was created by the XYZ Bus Driver that supports a bus specific driver override mechanism.

![Child Device Handle](image)

\textbf{OM13154}

\textbf{Figure 2-13 Child Device Handle with a Bus Specific Override}

\subsection*{2.5.6 Platform Components}

Under the UEFI Driver Model, the act of connecting and disconnecting drivers from controllers in a platform is under the platform firmware’s control. This will typically be implemented as part of the UEFI Boot Manager, but other implementations are possible. The boot services \texttt{EFI_BOOT_SERVICES.ConnectController()} and \texttt{EFI_BOOT_SERVICES.DisconnectController()} can be used by the platform firmware to determine which controllers get started and which ones do not. If the platform wishes to perform system diagnostics or install an operating system, then it may choose to connect drivers to all possible boot devices. If a platform wishes to boot a preinstalled operating system, it may choose to only connect drivers to the devices that are required to boot the selected operating system. The UEFI Driver Model supports both these modes of operation through the boot services \texttt{ConnectController()} and \texttt{DisconnectController()}. In addition, since the platform component that is in charge of booting the
platform has to work with device paths for console devices and boot options, all of the services and protocols involved in the UEFI Driver Model are optimized with device paths in mind.

Since the platform firmware may choose to only connect the devices required to produce consoles and gain access to a boot device, the OS present device drivers cannot assume that a UEFI driver for a device has been executed. The presence of a UEFI driver in the system firmware or in an option ROM does not guarantee that the UEFI driver will be loaded, executed, or allowed to manage any devices in a platform. All OS present device drivers must be able to handle devices that have been managed by a UEFI driver and devices that have not been managed by an UEFI driver.

The platform may also choose to produce a protocol named the Platform Driver Override Protocol. This is similar to the Bus Specific Driver Override Protocol, but it has higher priority. This gives the platform firmware the highest priority when deciding which drivers are connected to which controllers. The Platform Driver Override Protocol is attached to a handle in the system. The boot service ConnectController() will make use of this protocol if it is present in the system.

2.5.7 Hot-Plug Events

In the past, system firmware has not had to deal with hot-plug events in the preboot environment. However, with the advent of buses like USB, where the end user can add and remove devices at any time, it is important to make sure that it is possible to describe these types of buses in the UEFI Driver Model. It is up to the bus driver of a bus that supports the hot adding and removing of devices to provide support for such events. For these types of buses, some of the platform management is going to have to move into the bus drivers. For example, when a keyboard is hot added to a USB bus on a platform, the end user would expect the keyboard to be active. A USB Bus driver could detect the hot-add event and create a child handle for the keyboard device. However, because drivers are not connected to controllers unless EFI_BOOT_SERVICES.ConnectController() is called, the keyboard would not become an active input device. Making the keyboard driver active requires the USB Bus driver to call ConnectController() when a hot-add event occurs. In addition, the USB Bus Driver would have to call EFI_BOOT_SERVICES.DisconnectController() when a hot-remove event occurs. If EFI_BOOT_SERVICES.DisconnectController() returns an error the USB Bus Driver needs to retry the EFI_BOOT_SERVICES.DisconnectController() from a timer event until it succeeds.

Device drivers are also affected by these hot-plug events. In the case of USB, a device can be removed without any notice. This means that the Stop() functions of USB device drivers will have to deal with shutting down a driver for a device that is no longer present in the system. As a result, any outstanding I/O requests will have to be flushed without actually being able to touch the device hardware.

In general, adding support for hot-plug events greatly increases the complexity of both bus drivers and device drivers. Adding this support is up to the driver writer, so the extra complexity and size of the driver will need to be weighed against the need for the feature in the preboot environment.

2.5.8 EFI Services Binding

The UEFI Driver Model maps well onto hardware devices, hardware bus controllers, and simple combinations of software services that layer on top of hardware devices. However, the UEFI driver Model does not map well onto complex combinations of software services. As a result, an additional set of complementary protocols are required for more complex combinations of software services.

Figure 2-14 contains three examples showing the different ways that software services relate to each other. In the first two cases, each service consumes one or more other services, and at most one other
service consumes all of the services. Case #3 differs because two different services consume service A. The `EFI_DRIVER_BINDING_PROTOCOL` can be used to model cases #1 and #2, but it cannot be used to model case #3 because of the way that the UEFI Boot Service `OpenProtocol()` behaves. When used with the `BY_DRIVER` open mode, `OpenProtocol()` allows each protocol to have only at most one consumer. This feature is very useful and prevents multiple drivers from attempting to manage the same controller. However, it makes it difficult to produce sets of software services that look like case #3.

![Figure 2-14 Software Service Relationships](image)

The `EFI_SERVICE_BINDING_PROTOCOL` provides the mechanism that allows protocols to have more than one consumer. The `EFI_SERVICE_BINDING_PROTOCOL` is used with the `EFI_DRIVER_BINDING_PROTOCOL`. A UEFI driver that produces protocols that need to be available to more than one consumer at the same time will produce both the `EFI_DRIVER_BINDING_PROTOCOL` and the `EFI_SERVICE_BINDING_PROTOCOL`. This type of driver is a hybrid driver that will produce the `EFI_DRIVER_BINDING_PROTOCOL` in its driver entry point.

When the driver receives a request to start managing a controller, it will produce the `EFI_SERVICE_BINDING_PROTOCOL` on the handle of the controller that is being started. The `EFI_SERVICE_BINDING_PROTOCOL` is slightly different from other protocols defined in the UEFI Specification. It does not have a GUID associated with it. Instead, this protocol instance structure actually represents a family of protocols. Each software service driver that requires an `EFI_SERVICE_BINDING_PROTOCOL` instance will be required to generate a new GUID for its own type of `EFI_SERVICE_BINDING_PROTOCOL`. This requirement is why the various network protocols in this specification contain two GUIDs. One is the `EFI_SERVICE_BINDING_PROTOCOL` GUID for that network protocol, and the other GUID is for the protocol that contains the specific member services produced by the network driver. The mechanism defined here is not limited to network protocol drivers. It can be applied to any set of protocols that the `EFI_DRIVER_BINDING_PROTOCOL` cannot directly map because the protocols contain one or more relationships like case #3 in Figure 2-14.

Neither the `EFI_DRIVER_BINDING_PROTOCOL` nor the combination of the `EFI_DRIVER_BINDING_PROTOCOL` and the `EFI_SERVICE_BINDING_PROTOCOL` can handle circular
dependencies. There are methods to allow circular references, but they require that the circular link be present for short periods of time. When the protocols across the circular link are used, these methods also require that the protocol must be opened with an open mode of **EXCLUSIVE**, so that any attempts to deconstruct the set of protocols with a call to **DisconnectController()** will fail. As soon as the driver is finished with the protocol across the circular link, the protocol should be closed.

### 2.6 Requirements

This document is an architectural specification. As such, care has been taken to specify architecture in ways that allow maximum flexibility in implementation. However, there are certain requirements on which elements of this specification must be implemented to ensure that operating system loaders and other code designed to run with UEFI boot services can rely upon a consistent environment.

For the purposes of describing these requirements, the specification is broken up into required and optional elements. In general, an optional element is completely defined in the section that matches the element name. For required elements however, the definition may in a few cases not be entirely self contained in the section that is named for the particular element. In implementing required elements, care should be taken to cover all the semantics defined in this specification that relate to the particular element.

#### 2.6.1 Required Elements

**Table 2-11** lists the required elements. Any system that is designed to conform to this specification **must** provide a complete implementation of all these elements. This means that all the required service functions and protocols must be present and the implementation must deliver the full semantics defined in the specification for all combinations of calls and parameters. Implementers of applications, drivers or operating system loaders that are designed to run on a broad range of systems conforming to the UEFI specification may assume that all such systems implement all the required elements.

A system vendor may choose not to implement all the required elements, for example on specialized system configurations that do not support all the services and functionality implied by the required elements. However, since most applications, drivers and operating system loaders are written assuming all the required elements are present on a system that implements the UEFI specification; any such code is likely to require explicit customization to run on a less than complete implementation of the required elements in this specification.

**Table 2-11 Required UEFI Implementation Elements**

<table>
<thead>
<tr>
<th>Element</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>EFI_SYSTEM_TABLE</td>
<td>Provides access to UEFI Boot Services, UEFI Runtime Services, consoles, firmware vendor information, and the system configuration tables.</td>
</tr>
<tr>
<td>EFI_BOOT_SERVICES</td>
<td>All functions defined as boot services.</td>
</tr>
<tr>
<td>EFI_RUNTIME_SERVICES</td>
<td>All functions defined as runtime services.</td>
</tr>
<tr>
<td>EFI_LOADED_IMAGE_PROTOCOL</td>
<td>Provides information on the image.</td>
</tr>
<tr>
<td>EFI_LOADED_IMAGE_DEVICE_PATH_PROTOCOL</td>
<td>Specifies the device path that was used when a PE/COFF image was loaded through the EFI Boot Service LoadImage().</td>
</tr>
<tr>
<td>EFIDEVICE_PATH_PROTOCOL</td>
<td>Provides the location of the device.</td>
</tr>
</tbody>
</table>
2.6.2 Platform-Specific Elements

There are a number of elements that can be added or removed depending on the specific features that a platform requires. Platform firmware developers are required to implement UEFI elements based upon the features included. The following is a list of potential platform features and the elements that are required for each feature type:

1. If a platform includes console devices, the EFI_SIMPLE_TEXT_INPUT_PROTOCOL, EFI_SIMPLE_TEXT_INPUT_EX_PROTOCOL, and EFI_SIMPLE_TEXT_OUTPUT_PROTOCOL must be implemented.

2. If a platform includes a configuration infrastructure, then the EFI_HII_DATABASE_PROTOCOL, EFI_HII_STRING_PROTOCOL, EFI_HII_CONFIG_ROUTING_PROTOCOL, and EFI_HII_CONFIG_ACCESS_PROTOCOL are required. If you support bitmapped fonts, you must support EFI_HII_FONT_PROTOCOL.

3. If a platform includes graphical console devices, then the EFI_GRAPHICS_OUTPUT_PROTOCOL, EFI_EDID_DISCOVERED_PROTOCOL, and EFI_EDID_ACTIVE_PROTOCOL must be implemented. In order to support the EFI_GRAPHICS_OUTPUT_PROTOCOL, a platform must contain a driver to consume EFI_GRAPHICS_OUTPUT_PROTOCOL and produce EFI_SIMPLE_TEXT_OUTPUT_PROTOCOL even if the EFI_GRAPHICS_OUTPUT_PROTOCOL is produced by an external driver.

4. If a platform includes a pointer device as part of its console support, the EFI_SIMPLE_POINTER_PROTOCOL must be implemented.

5. If a platform includes the ability to boot from a disk device, then the EFI_BLOCK_IO_PROTOCOL, the EFI_DISK_IO_PROTOCOL, the EFI_SIMPLE_FILE_SYSTEM_PROTOCOL, and the EFI_UNICODE_COLLATION_PROTOCOL are required. In addition, partition support for MBR, GPT, and El Torito must be implemented. For disk devices supporting the security commands of the SPC-4 or ATA8-ACS command set, the EFI_STORAGE_SECURITY_COMMAND_PROTOCOL is also required. An external driver may produce the Block I/O Protocol and the EFI_STORAGE_SECURITY_COMMAND_PROTOCOL. All other protocols required to boot from a disk device must be carried as part of the platform.

6. If a platform includes the ability to perform a TFTP-based boot from a network device, then the EFI_PXE_BASE_CODE_PROTOCOL is required. The platform must be prepared to produce this protocol on any of EFI_NETWORK_INTERFACE_IDENTIFIER_PROTOCOL (UNDI), EFI_SIMPLE_NETWORK_PROTOCOL, or the EFI_MANAGED_NETWORK_PROTOCOL. If a platform includes the ability to validate a boot image received through a network device, it is also required that image verification be supported, including SetupMode equal zero and the boot image hash or a verification certificate corresponding to the image exist in the 'db' variable and not in the 'dbx' variable. An external driver may produce the UNDI interface. All other protocols required to boot from a network device must be carried by the platform.

7. If a platform supports UEFI general purpose network applications, then the EFI_MANAGED_NETWORK_PROTOCOL, EFI_MANAGED_NETWORK_SERVICE_BINDING_PROTOCOL, EFI_ARP_PROTOCOL, EFI_ARP_SERVICE_BINDING_PROTOCOL, EFI_DHCP4_PROTOCOL, EFI_DHCP4_SERVICE_BINDING_PROTOCOL, EFI_TCP4_PROTOCOL, EFI_TCP4_SERVICE_BINDING_PROTOCOL, EFI_IP4_PROTOCOL, EFI_IP4_SERVICE_BINDING_PROTOCOL, and EFI_IP4_CONFIG2_PROTOCOL.
EFI_UDP4_PROTOCOL, and EFI_UDP4_SERVICE_BINDING_PROTOCOL are required. If additional IPv6 support is needed for the platform, then EFI_DHCP6_PROTOCOL, EFI_DHCP6_SERVICE_BINDING_PROTOCOL, EFI_TCP6_PROTOCOL, EFI_TCP6_SERVICE_BINDING_PROTOCOL, EFI_IP6_PROTOCOL, EFI_IP6_SERVICE_BINDING_PROTOCOL, EFI_IP6_CONFIG_PROTOCOL, EFI_UDP6_PROTOCOL, and EFI_UDP6_SERVICE_BINDING_PROTOCOL are additionally required. If the network application requires DNS capability, EFI_DNS4_SERVICE_BINDING_PROTOCOL and EFI_DNS4_PROTOCOL are required for the IPv4 stack. EFI_DNS6_SERVICE_BINDING_PROTOCOL and EFI_DNS6_PROTOCOL are required for the IPv6 stack. If the network environment requires TLS feature, EFI_TLS_SERVICE_BINDING_PROTOCOL and EFI_TLS_PROTOCOL are required. If the network environment requires IPSEC feature, EFI_IPSEC_CONFIG_PROTOCOL and EFI_IPSEC2_PROTOCOL are required. If the network environment requires VLAN features, EFI_VLAN_CONFIG_PROTOCOL is required.

8. If a platform includes a byte-stream device such as a UART, then the EFI_SERIAL_IO_PROTOCOL must be implemented.

9. If a platform includes PCI bus support, then the EFI_PCI_ROOT_BRIDGE_IO_PROTOCOL, the EFI_PCI_IO_PROTOCOL, must be implemented.

10. If a platform includes USB bus support, then the EFI_USB2_HC_PROTOCOL and the EFI_USB_IO_PROTOCOL must be implemented. An external device can support USB by producing a USB Host Controller Protocol.

11. If a platform includes an NVM Express controller, then the EFI_NVM_EXPRESS_PASS_THRU_PROTOCOL must be implemented.

12. If a platform supports booting from a block-oriented NVM Express controller, then the EFI_BLOCK_IO_PROTOCOL must be implemented. An external driver may produce the EFI_NVM_EXPRESS_PASS_THRU_PROTOCOL. All other protocols required to boot from an NVM Express subsystem must be carried by the platform.

13. If a platform includes an I/O subsystem that utilizes SCSI command packets, then the EFI_EXT_SCSI_PASS_THRU_PROTOCOL must be implemented.

14. If a platform supports booting from a block oriented SCSI peripheral, then the EFI_SCSI_IO_PROTOCOL and EFI_BLOCK_IO_PROTOCOL must be implemented. An external driver may produce the EFI_EXT_SCSI_PASS_THRU_PROTOCOL. All other protocols required to boot from a SCSI I/O subsystem must be carried by the platform.

15. If a platform supports booting from an iSCSI peripheral, then the EFI_ISCSI_INITIATOR_NAME_PROTOCOL and the EFI_AUTHENTICATION_INFO_PROTOCOL must be implemented.

16. If a platform includes debugging capabilities, then the EFI_DEBUG_SUPPORT_PROTOCOL, the EFI_DEBUGPORT_PROTOCOL, and the EFI_IMAGE_INFO Table must be implemented.

17. If a platform includes the ability to override the default driver to the controller matching algorithm provided by the UEFI Driver Model, then the EFI_PLATFORM_DRIVER_OVERRIDE_PROTOCOL must be implemented.

18. If a platform includes an I/O subsystem that utilizes ATA command packets, then the EFIATA_PASS_THRU_PROTOCOL must be implemented.

19. If a platform supports option ROMs from devices not permanently attached to the platform and it supports the ability to authenticate those option ROMs, then it must support the option ROM validation methods described in Network Protocols — UDP and MTFTP and the authenticated EFI variables described in Section 8.11.

20. If a platform includes the ability to authenticate UEFI images and the platform potentially supports more than one OS loader, it must support the methods described in Network Protocols — UDP and MTFTP and the authenticated UEFI variables described in Section 8.11.

21. EBC support is no longer required as of UEFI Specification version 2.8. If an EBC interpreter is implemented, then it must produce the EFI_EBC_PROTOCOL interface.
22. If a platform includes the ability to perform a HTTP-based boot from a network device, then the
   EFI HTTP SERVICE BINDING_PROTOCOL, EFI HTTP_PROTOCOL and
   EFI HTTP UTILITIES_PROTOCOL are required. If it includes the ability to perform a HTTPS-based
   boot from network device, besides above protocols, EFI TLS SERVICE BINDING_PROTOCOL,
   EFI TLS_PROTOCOL and EFI TLS_CONFIGURATION_PROTOCOL are also required. If it includes
   the ability to perform a HTTP(S)-based boot with DNS feature, then
   EFI DNS4 SERVICE BINDING_PROTOCOL, EFI DNS4_PROTOCOL and EFI DNS6_PROTOCOL are required for the IPv4
   stack; EFI DNS6_SERVICE BINDING_PROTOCOL and EFI DNS6_PROTOCOL are required for the
   IPv6 stack.

23. If a platform includes the ability to perform a wireless boot from a network device with EAP feature, and
   if this platform provides a standalone wireless EAP driver, then EFI EAP_PROTOCOL,
   EFI EAP_CONFIGURATION_PROTOCOL, and EFI EAP_MANAGEMENT2_PROTOCOL are
   required; if the platform provides a standalone wireless supplicant, then
   EFI SUPPLICANT_PROTOCOL and EFI EAP_CONFIGURATION_PROTOCOL are required. If it
   includes the ability to perform a wireless boot with TLS feature, then
   EFI TLS SERVICE BINDING_PROTOCOL, EFI TLS_PROTOCOL and
   EFI_TLS_CONFIGURATION_PROTOCOL are required.

24. If a platform supports classic Bluetooth, then EFI BLUETOOTH_HC_PROTOCOL,
   EFI BLUETOOTH_IO_PROTOCOL, and EFI BLUETOOTH_CONFIG_PROTOCOL must be
   implemented, and EFI BLUETOOTH ATTRIBUTE_PROTOCOL may be implemented. If a platform
   supports Bluetooth Smart (Bluetooth Low Energy), then EFI BLUETOOTH_HC_PROTOCOL,
   EFI BLUETOOTH_ATTRIBUTE_PROTOCOL and EFI BLUETOOTH LE CONFIG_PROTOCOL
   must be implemented. If a platform supports both Bluetooth classic and BluetoothLE, then both above
   requirements should be satisfied.

25. If a platform supports RESTful communication over HTTP or over an in-band path to a BMC, then the
   EFI REST_PROTOCOL or EFI REST_EX_PROTOCOL must be implemented. If
   EFI REST_EX_PROTOCOL is implemented, EFI REST_EX_SERVICE BINDING_PROTOCOL
   must be implemented as well. If a platform supports Redfish communication over HTTP or over an in-band
   path to a BMC, the EFI REDFISH_DISCOVER_PROTOCOL and
   EFI REST_JSON_STRUCTURE_PROTOCOL may be implemented.

26. If a platform includes the ability to use a hardware feature to create high quality random numbers, this
   capability should be exposed by instance of EFI_RNG_PROTOCOL with at least one EFI RNG Algorithm
   supported.

27. If a platform permits the installation of Load Option Variables, (Boot####, or Driver####, or
    SysPrep####), the platform must support and recognize all defined values for Attributes within the
    variable and report these capabilities in BootOptionSupport. If a platform supports installation of Load
    Option Variables of type Driver####, all installed Driver#### variables must be processed and the
    indicated driver loaded and initialized during every boot. And all installed SysPrep#### options must be
    processed prior to processing Boot#### options.

28. If the platform supports UEFI secure boot as described in Secure Boot and Driver Signing, the platform
    must provide the PKCS verification functions described in Section 37.4.

29. If a platform includes an I/O subsystem that utilizes SD or eMMC command packets, then the
    EFI_SD_MMC_PASS_THRU_PROTOCOL must be implemented.

30. If a platform includes the ability to create/destroy a specified RAM disk, the
    EFI_RAM_DISK_PROTOCOL must be implemented and only one instance of this protocol exists.

31. If a platform includes a mass storage device which supports hardware-based erase on a specified range,
    then the EFI ERASE BLOCK_PROTOCOL must be implemented.

32. If a platform includes the ability to register for notifications when a call to ResetSystem is called, then
    the EFI_RESET_NOTIFICATION_PROTOCOL must be implemented.

33. If a platform includes UFS devices, the EFI_UFS_DEVICE_CONFIG_PROTOCOL must be
    implemented.
34. If a platform cannot support calls defined in `EFI_RUNTIME_SERVICES` after `ExitBootServices()` is called, that platform is permitted to provide implementations of those runtime services that return `EFI_UNSUPPORTED` when invoked at runtime. On such systems, an `EFI_RT_PROPERTIES_TABLE` configuration table should be published describing which runtime services are supported at runtime.

35. If a platform includes support for CXL devices with coherent memory, then the platform must support extracting the Coherent Device Attribute Table (CDAT) from the device, using either the CXL Data Object Exchange services (as defined in the CXL 2.0 Specification) or the `EFI_ADAPTER_INFORMATION_PROTOCOL` instance (with `EFI_ADAPTER_INFO_CDAT_TYPE_GUID` type) installed on that device.

**Note:** Some of the required protocol instances are created by the corresponding Service Binding Protocol. For example, `EFI_IP4_PROTOCOL` is created by `EFI_IP4_SERVICE_BINDING_PROTOCOL`. Please refer to the corresponding sections of Service Binding Protocol for the details.

### 2.6.3 Driver-Specific Elements

There are a number of UEFI elements that can be added or removed depending on the features that a specific driver requires. Drivers can be implemented by platform firmware developers to support buses and devices in a specific platform. Drivers can also be implemented by add-in card vendors for devices that might be integrated into the platform hardware or added to a platform through an expansion slot.

The following list includes possible driver features, and the UEFI elements that are required for each feature type:

1. If a driver follows the driver model of this specification, the `EFI_DRIVER_BINDING_PROTOCOL` must be implemented. It is strongly recommended that all drivers that follow the driver model of this specification also implement the `EFI_COMPONENT_NAME2_PROTOCOL`.
2. If a driver requires configuration information, the driver must use the `EFI_HII_DATABASE_PROTOCOL`. A driver should not otherwise display information to the user or request information from the user.
3. If a driver requires diagnostics, the `EFI_DRIVER_DIAGNOSTICS2_PROTOCOL` must be implemented. In order to support low boot times, limit diagnostics during normal boots. Time consuming diagnostics should be deferred until the `EFI_DRIVER_DIAGNOSTICS2_PROTOCOL` is invoked.
4. If a bus supports devices that are able to provide containers for drivers (e.g. option ROMs), then the bus driver for that bus type must implement the `EFI_BUS_SPECIFIC_DRIVER_OVERRIDE_PROTOCOL`.
5. If a driver is written for a console output device, then the `EFI_SIMPLE_TEXT_OUTPUT_PROTOCOL` must be implemented.
6. If a driver is written for a graphical console output device, then the `EFI_GRAPHICS_OUTPUT_PROTOCOL`, `EFI_EDID_DISCOVERED_PROTOCOL` and `EFI_EDID_ACTIVE_PROTOCOL` must be implemented.
7. If a driver is written for a console input device, then the `EFI_SIMPLE_TEXT_INPUT_PROTOCOL` and `EFI_SIMPLE_TEXT_INPUT_EX_PROTOCOL` must be implemented.
8. If a driver is written for a pointer device, then the `EFI_SIMPLE_POINTER_PROTOCOL` must be implemented.
9. If a driver is written for a network device, then the `EFI_NETWORK_INTERFACE_IDENTIFIER_PROTOCOL`, `EFI_SIMPLE_NETWORK_PROTOCOL` or `EFI_MANAGED_NETWORK_PROTOCOL` must be implemented. If VLAN is supported in hardware, then driver for the network device may implement the `EFI_VLAN_CONFIG_PROTOCOL`. If a network device chooses to only produce the `EFI_MANAGED_NETWORK_PROTOCOL`, then the driver for the network device must implement the `EFI_VLAN_CONFIG_PROTOCOL`. If a driver is written for a network device to supply wireless feature, besides above protocols,
EFI_ADAPTER_INFORMATION_PROTOCOL must be implemented. If the wireless driver does not provide user configuration capability, EFI_WIRELESS_MAC_CONNECTION_I_PROTOCOL must be implemented. If the wireless driver is written for a platform which provides a standalone wireless EAP driver, EFI_EAP_PROTOCOL must be implemented.

10. If a driver is written for a disk device, then the EFI_BLOCK_IO_PROTOCOL and the EFI_BLOCK_IO2_PROTOCOL must be implemented. In addition, the EFI_STORAGE_SECURITY_COMMAND_PROTOCOL must be implemented for disk devices supporting the security commands of the SPC-4 or ATA8-ACS command set. In addition, for devices that support incline encryption in the host storage controller, the EFI_BLOCK_IO_CRYPTO_PROTOCOL must be supported.

11. If a driver is written for a disk device, then the EFI_BLOCK_IO_PROTOCOL and the EFI_BLOCK_IO2_PROTOCOL must be implemented. In addition, the EFI_STORAGE_SECURITY_COMMAND_PROTOCOL must be implemented for disk devices supporting the security commands of the SPC-4 or ATA8-ACS command set.

12. If a driver is written for a device that is not a block oriented device but one that can provide a file system-like interface, then the EFI_SIMPLE_FILE_SYSTEM_PROTOCOL must be implemented.

13. If a driver is written for a PCI root bridge, then the EFI_PCI_ROOT_BRIDGE_IO_PROTOCOL and the EFI_PCI_IO_PROTOCOL must be implemented.

14. If a driver is written for an NVM Express controller, then the EFI_NVM_EXPRESS_PASS_THRU_PROTOCOL must be implemented.

15. If a driver is written for a USB host controller, then the EFI_USB2_HC_PROTOCOL and the EFI_USB_IO_PROTOCOL must be implemented. If a driver is written for a USB host controller, then the must be implemented.

16. If a driver is written for a SCSI controller, then the EFI_EXT_SCSI_PASS_THRU_PROTOCOL must be implemented.

17. If a driver is digitally signed, it must embed the digital signature in the PE/COFF image as described in "Embedded Signatures" on page 1691.

18. If a driver is written for a boot device that is not a block-oriented device, a file system-based device, or a console device, then the EFI_LOAD_FILE2_PROTOCOL must be implemented.

19. If a driver follows the driver model of this specification, and the driver wants to produce warning or error messages for the user, then the EFI_DRIVER_HEALTH_PROTOCOL must be used to produce those messages. The Boot Manager may optionally display the messages to the user.

20. If a driver follows the driver model of this specification, and the driver needs to perform a repair operation that is not part of the normal initialization sequence, and that repair operation requires an extended period of time, then the EFI_DRIVER_HEALTH_PROTOCOL must be used to provide the repair feature. If the Boot Manager detects a boot device that requires a repair operation, then the Boot Manager must use the EFI_DRIVER_HEALTH_PROTOCOL to perform the repair operation. The Boot Manager can optionally display progress indicators as the repair operation is performed by the driver.

21. If a driver follows the driver model of this specification, and the driver requires the user to make software and/or hardware configuration changes before the boot devices that the driver manages can be used, then the EFI_DRIVER_ATTACH_PROTOCOL must be used. If the Boot Manager detects a boot device that requires software and/or hardware configuration changes to make the boot device usable, then the Boot Manager may optionally allow the user to make those configuration changes.

22. If a driver is written for an ATA controller, then the EFI_ATA_PASS_THRU_PROTOCOL must be implemented.

23. If a driver follows the driver model of this specification, and the driver wants to be used with higher priority than the Bus Specific Driver Override Protocol when selecting the best driver for controller, then the EFI_DRIVER_FAMILY_OVERRIDE_PROTOCOL must be produced on the same handle as the EFI_DRIVER_BINDING_PROTOCOL.

24. If a driver supports firmware management by an external agent or application, then the EFI_FIRMWARE_MANAGEMENT_PROTOCOL must be used to support firmware management.
25. If a driver follows the driver model of this specification and a driver is a device driver as defined in Section 2.5, it must perform bus transactions via the bus abstraction protocol produced by a parent bus driver. Thus a driver for a device that conforms to the PCI specification must use EFI_PCI_IO_PROTOCOL for all PCI memory space, PCI I/O, PCI configuration space, and DMA operations.

26. If a driver is written for a classic Bluetooth controller, then EFI_BLUETOOTH_HC_PROTOCOL, EFI_BLUETOOTH_IO_PROTOCOL and EFI_BLUETOOTH_CONFIG_PROTOCOL must be implemented, and EFI_BLUETOOTH_ATTRIBUTE_PROTOCOL may be implemented. If a driver written for a Bluetooth Smart (Bluetooth Low Energy) controller, then EFI_BLUETOOTH_HC_PROTOCOL, EFI_BLUETOOTH_ATTRIBUTE_PROTOCOL and EFI_BLUETOOTH_LE_CONFIG_PROTOCOL must be implemented. If a driver supports both Bluetooth classic and BluetoothLE, then both above requirements should be satisfied.

27. If a driver is written for an SD controller or eMMC controller, then the EFI_SD_MMC_PASS_THRU_PROTOCOL must be implemented.

28. If a driver is written for a UFS device, then EFI_UFS_DEVICE_CONFIG_PROTOCOL must be implemented.

2.6.4 Extensions to this Specification published elsewhere

This specification has been extended over time to include support for new devices and technologies. As the name of the specification implies, the original intent in its definition was to create a baseline for firmware interfaces that is extensible without the need to include extensions in the main body of this specification.

Readers of this specification may find that a feature or type of device is not treated by the specification. This does not necessarily mean that there is no agreed "standard" way to support the feature or device in implementations that claim conformance to this Specification. On occasion, it may be more appropriate for other standards organizations to publish their own extensions that are designed to be used in concert with the definitions presented here. This may for example allow support for new features in a more timely fashion than would be accomplished by waiting for a revision to this specification or perhaps that such support is defined by a group with a specific expertise in the subject area. Readers looking for means to access features or devices that are not treated in this document are therefore recommended to inquire of appropriate standards groups to ascertain if appropriate extension publications already exist before creating their own extensions.

By way of examples, at the time of writing the UEFI Forum is aware of a number of extension publications that are compatible with and designed for use with this specification. Such extensions include:

Developers Interface Guide for Itanium® Architecture Based Servers: published and hosted by the DIG64 group (See “Links to UEFI-Related Documents” (http://uefi.org/uefi) under the heading “Developers Interface Guide for Itanium® Architecture Based Servers”). This document is a set of technical guidelines that define hardware, firmware, and operating system compatibility for Itanium™-based servers;

TCG EFI Platform Specification: published and hosted by the Trusted Computing Group (See “Links to UEFI-Related Documents” (http://uefi.org/uefi) under the heading “TCG EFI Platform Specification”). This document is about the processes that boot an EFI platform and boot an OS on that platform. Specifically, this specification contains the requirements for measuring boot events into TPM PCRs and adding boot event entries into the Event Log.

TCG EFI Protocol Specification: published and hosted by the Trusted Computing Group (See “Links to UEFI-Related Documents” (http://uefi.org/uefi) under the
heading “TCG EFI Protocol Specification”). This document defines a standard interface to the TPM on an EFI platform.

Other extension documents may exist outside the view of the UEFI Forum or may have been created since the last revision of this document.
3 - Boot Manager

The UEFI boot manager is a firmware policy engine that can be configured by modifying architecturally defined global NVRAM variables. The boot manager will attempt to load UEFI drivers and UEFI applications (including UEFI OS boot loaders) in an order defined by the global NVRAM variables. The platform firmware must use the boot order specified in the global NVRAM variables for normal boot. The platform firmware may add extra boot options or remove invalid boot options from the boot order list.

The platform firmware may also implement value added features in the boot manager if an exceptional condition is discovered in the firmware boot process. One example of a value added feature would be not loading a UEFI driver if booting failed the first time the driver was loaded. Another example would be booting to an OEM-defined diagnostic environment if a critical error was discovered in the boot process.

The boot sequence for UEFI consists of the following:

- The boot order list is read from a globally defined NVRAM variable. Modifications to this variable are only guaranteed to take effect after the next platform reset. The boot order list defines a list of NVRAM variables that contain information about what is to be booted. Each NVRAM variable defines a name for the boot option that can be displayed to a user.
- The variable also contains a pointer to the hardware device and to a file on that hardware device that contains the UEFI image to be loaded.
- The variable might also contain paths to the OS partition and directory along with other configuration specific directories.

The NVRAM can also contain load options that are passed directly to the UEFI image. The platform firmware has no knowledge of what is contained in the load options. The load options are set by higher level software when it writes to a global NVRAM variable to set the platform firmware boot policy. This information could be used to define the location of the OS kernel if it was different than the location of the UEFI OS loader.

3.1 Firmware Boot Manager

The boot manager is a component in firmware conforming to this specification that determines which drivers and applications should be explicitly loaded and when. Once compliant firmware is initialized, it passes control to the boot manager. The boot manager is then responsible for determining what to load and any interactions with the user that may be required to make such a decision.

The actions taken by the boot manager depend upon the system type and the policies set by the system designer. For systems that allow the installation of new Boot Variables (Section 3.4), the Boot Manager must automatically or upon the request of the loaded item, initialize at least one system console, as well as perform all required initialization of the device indicated within the primary boot target. For such systems, the Boot Manager is also required to honor the priorities set in BootOrder variable.

In particular, likely implementation options might include any console interface concerning boot, integrated platform management of boot selections, and possible knowledge of other internal applications or recovery drivers that may be integrated into the system through the boot manager.
3.1.1 Boot Manager Programming

Programmatic interaction with the boot manager is accomplished through globally defined variables. On initialization the boot manager reads the values which comprise all of the published load options among the UEFI environment variables. By using the `SetVariable()` function the data that contain these environment variables can be modified. Such modifications are guaranteed to take effect after the next system boot commences. However, boot manager implementations may choose to improve on this guarantee and have changes take immediate effect for all subsequent accesses to the variables that affect boot manager behavior without requiring any form of system reset.

Each load option entry resides in a `Boot####`, `Driver####`, `SysPrep####`, `OsRecovery####` or `PlatformRecovery####` variable where `####` is replaced by a unique option number in printable hexadecimal representation using the digits 0–9, and the upper case versions of the characters A–F (0000–FFFF).

The `####` must always be four digits, so small numbers must use leading zeros. The load options are then logically ordered by an array of option numbers listed in the desired order. There are two such option ordering lists when booting normally. The first is `DriverOrder` that orders the `Driver####` load option variables into their load order. The second is `BootOrder` that orders the `Boot####` load option variables into their load order.

For example, to add a new boot option, a new `Boot####` variable would be added. Then the option number of the new `Boot####` variable would be added to the `BootOrder` ordered list and the `BootOrder` variable would be rewritten. To change boot option on an existing `Boot####`, only the `Boot####` variable would need to be rewritten. A similar operation would be done to add, remove, or modify the driver load list.

If the boot via `Boot####` returns with a status of `EFI_SUCCESS`, platform firmware supports boot manager menu, and if firmware is configured to boot in an interactive mode, the boot manager will stop processing the `BootOrder` variable and present a boot manager menu to the user. If any of the above-mentioned conditions is not satisfied, the next `Boot####` in the `BootOrder` variable will be tried until all possibilities are exhausted. In this case, boot option recovery must be performed (see Section 3.4).

The boot manager may perform automatic maintenance of the database variables. For example, it may remove unreferenced load option variables or any load option variables that cannot be parsed, and it may rewrite any ordered list to remove any load options that do not have corresponding load option variables. The boot manager can also, at its own discretion, provide an administrator with the ability to invoke manual maintenance operations as well. Examples include choosing the order of any or all load options, activating or deactivating load options, initiating OS-defined or platform-defined recovery, etc. In addition, if a platform intends to create `PlatformRecovery####`, before attempting to load and execute any `DriverOrder` or `BootOrder` entries, the firmware must create any and all `PlatformRecovery####` variables (see Section 3.4.2). The firmware should not, under normal operation, automatically remove any correctly formed `Boot####` variable currently referenced by the `BootOrder` or `BootNext` variables. Such removal should be limited to scenarios where the firmware is guided by direct user interaction.

The contents of `PlatformRecovery####` represent the final recovery options the firmware would have attempted had recovery been initiated during the current boot, and need not include entries to reflect contingencies such as significant hardware reconfiguration, or entries corresponding to specific hardware that the firmware is not yet aware of.
The behavior of the UEFI Boot Manager is impacted when Secure Boot is enabled, See Section 32.4.

3.1.2 Load Option Processing

The boot manager is required to process the Driver load option entries before the Boot load option entries. If the EFI_OS_INDICATIONS_START_OS_RECOVERY bit has been set in OsIndications, the firmware shall attempt OS-defined recovery (see Section 3.4.1) rather than normal boot processing. If the EFI_OS_INDICATIONS_START_PLATFORM_RECOVERY bit has been set in OsIndications, the firmware shall attempt platform-defined recovery (see Section 3.4.2) rather than normal boot processing or handling of the EFI_OS_INDICATIONS_START_OS_RECOVERY bit. In either case, both bits should be cleared.

Otherwise, the boot manager is also required to initiate a boot of the boot option specified by the BootNext variable as the first boot option on the next boot, and only on the next boot. The boot manager removes the BootNext variable before transferring control to the BootNext boot option. After the BootNext boot option is tried, the normal BootOrder list is used. To prevent loops, the boot manager deletes BootNext before transferring control to the preselected boot option.

If all entries of BootNext and BootOrder have been exhausted without success, or if the firmware has been instructed to attempt boot order recovery, the firmware must attempt boot option recovery (see Section 3.4).

The boot manager must call EFI_BOOT_SERVICES.LoadImage() which supports at least EFI_SIMPLE_FILE_SYSTEM_PROTOCOL and EFI_LOAD_FILE_PROTOCOL for resolving load options. If LoadImage() succeeds, the boot manager must enable the watchdog timer for 5 minutes by using the EFI_BOOT_SERVICES.SetWatchdogTimer() boot service prior to calling EFI_BOOT_SERVICES.StartImage(). If a boot option returns control to the boot manager, the boot manager must disable the watchdog timer with an additional call to the SetWatchdogTimer() boot service.

If the boot image is not loaded via EFI_BOOT_SERVICES.LoadImage() the boot manager is required to check for a default application to boot. Searching for a default application to boot happens on both removable and fixed media types. This search occurs when the device path of the boot image listed in any boot option points directly to an EFI_SIMPLE_FILE_SYSTEM_PROTOCOL device and does not specify the exact file to load. The file discovery method is explained in Section 3.4. The default media boot case of a protocol other than EFI_SIMPLE_FILE_SYSTEM_PROTOCOL is handled by the EFI_LOAD_FILE_PROTOCOL for the target device path and does not need to be handled by the boot manager.

The UEFI boot manager must support booting from a short-form device path that starts with the first element being a USB WWID (see Table 10-23) or a USB Class (see Table 10-25) device path. For USB WWID, the boot manager must use the device vendor ID, device product id, and serial number, and must match any USB device in the system that contains this information. If more than one device matches the USB WWID device path, the boot manager will pick one arbitrarily. For USB Class, the boot manager must use the vendor ID, Product ID, Device Class, Device Subclass, and Device Protocol, and must match any USB device in the system that contains this information. If any of the ID, Product ID, Device Class, Device Subclass, or Device Protocol contain all F’s (0xFFFF or 0xFF), this element is skipped for the purpose of matching. If more than one device matches the USB Class device path, the boot manager will pick one arbitrarily.
The boot manager must also support booting from a short-form device path that starts with the first element being a hard drive media device path (see Table 10-49). The boot manager must use the GUID or signature and partition number in the hard drive device path to match it to a device in the system. If the drive supports the GPT partitioning scheme the GUID in the hard drive media device path is compared with the UniquePartitionGuid field of the GUID Partition Entry (see Table 5-6). If the drive supports the PC-AT MBR scheme the signature in the hard drive media device path is compared with the UniqueMBRSignature in the Legacy Master Boot Record (see Table 5-1). If a signature match is made, then the partition number must also be matched. The hard drive device path can be appended to the matching hardware device path and normal boot behavior can then be used. If more than one device matches the hard drive device path, the boot manager will pick one arbitrarily. Thus the operating system must ensure the uniqueness of the signatures on hard drives to guarantee deterministic boot behavior.

The boot manager must also support booting from a short-form device path that starts with the first element being a File Path Media Device Path (see Table 10-52). When the boot manager attempts to boot a short-form File Path Media Device Path, it will enumerate all removable media devices, followed by all fixed media devices, creating boot options for each device. The boot option FilePathList[0] is constructed by appending short-form File Path Media Device Path to the device path of a media. The order within each group is undefined. These new boot options must not be saved to non volatile storage, and may not be added to BootOrder. The boot manager will then attempt to boot from each boot option. If a device does not support the EFI_SIMPLE_FILE_SYSTEM_PROTOCOL, but supports the EFI_BLOCK_IO_PROTOCOL protocol, then the EFI Boot Service ConnectController must be called for this device with DriverImageHandle and RemainingDevicePath set to NULL and the Recursive flag is set to TRUE. The firmware will then attempt to boot from any child handles produced using the algorithms outlined above.

The boot manager must also support booting from a short-form device path that starts with the first element being a URI Device Path (see Table 10-40). When the boot manager attempts to boot a short-form URI Device Path, it could attempt to connect any device which will produce a device path protocol including a URI device path node until it matches a device, or fail to match any device. The boot manager will enumerate all LoadFile protocol instances, and invoke LoadFile protocol with FilePath set to the short-form device path during the matching process.

### 3.1.3 Load Options

Each load option variable contains an EFI_LOAD_OPTION descriptor that is a byte packed buffer of variable length fields.

```c
typedef struct _EFI_LOAD_OPTION {
    UINT32 Attributes;
    UINT16 FilePathListLength;
    // CHAR16 Description[];
    // EFI_DEVICE_PATH_PROTOCOL FilePathList[];
    // UINT8 OptionalData[];
} EFI_LOAD_OPTION;
```

**Parameters**

**Attributes**

The attributes for this load option entry. All unused bits must be zero and are reserved by the UEFI specification for future growth. See “Related Definitions.”
FilePathListLength
Length in bytes of the FilePathList. OptionalData starts at offset `sizeof(UINT32) + sizeof(UINT16) + StrSize(Description) + FilePathListLength` of the `EFI_LOAD_OPTION` descriptor.

Description
The user readable description for the load option. This field ends with a Null character.

FilePathList
A packed array of UEFI device paths. The first element of the array is a device path that describes the device and location of the Image for this load option. The `FilePathList[0]` is specific to the device type. Other device paths may optionally exist in the `FilePathList`, but their usage is OSV specific. Each element in the array is variable length, and ends at the device path end structure. Because the size of `Description` is arbitrary, this data structure is not guaranteed to be aligned on a natural boundary. This data structure may have to be copied to an aligned natural boundary before it is used.

OptionalData
The remaining bytes in the load option descriptor are a binary data buffer that is passed to the loaded image. If the field is zero bytes long, a `NULL` pointer is passed to the loaded image. The number of bytes in `OptionalData` can be computed by subtracting the starting offset of `OptionalData` from total size in bytes of the `EFI_LOAD_OPTION`.

Related Definitions
```c
//*******************************************************
// Attributes
//*******************************************************
#define LOAD_OPTION_ACTIVE        0x00000001
#define LOAD_OPTION_FORCE_RECONNECT 0x00000002
#define LOAD_OPTION_HIDDEN         0x00000008
#define LOAD_OPTION_CATEGORY       0x00001F00
#define LOAD_OPTION_CATEGORY_BOOT  0x00000000
#define LOAD_OPTION_CATEGORY_APP   0x00000100
// All values 0x00000200-0x00001F00 are reserved
```

Description
Calling `SetVariable()` creates a load option. The size of the load option is the same as the size of the `DataSize` argument to the `SetVariable()` call that created the variable. When creating a new load option, all undefined attribute bits must be written as zero. When updating a load option, all undefined attribute bits must be preserved.

If a load option is marked as `LOAD_OPTION_ACTIVE`, the boot manager will attempt to boot automatically using the device path information in the load option. This provides an easy way to disable or enable load options without needing to delete and re-add them.

If any `Driver###` load option is marked as `LOAD_OPTION_FORCE_RECONNECT`, then all of the UEFI drivers in the system will be disconnected and reconnected after the last `Driver###` load option is processed. This allows a UEFI driver loaded with a `Driver###` load option to override a UEFI driver that was loaded prior to the execution of the UEFI Boot Manager.
The executable indicated by FilePathList[0] in Driver#### load option must be of type EFI_IMAGE_SUBSYSTEM_EFI_BOOT_SERVICE_DRIVER or EFI_IMAGE_SUBSYSTEM_EFI_RUNTIME_DRIVER otherwise the indicated executable will not be entered for initialization.

The executable indicated by FilePathList[0] in SysPrep####, Boot####, or OsRecovery#### load option must be of type EFI_IMAGE_SUBSYSTEM_EFI_APPLICATION, otherwise the indicated executable will not be entered.

The LOAD_OPTION_CATEGORY is a sub-field of Attributes that provides details to the boot manager to describe how it should group the Boot#### load options. This field is ignored for variables of the form Driver####, SysPrep####, or OsRecovery####.

Boot#### load options with LOAD_OPTION_CATEGORY set to LOAD_OPTION_CATEGORY_BOOT are meant to be part of the normal boot processing.

Boot#### load options with LOAD_OPTION_CATEGORY set to LOAD_OPTION_CATEGORY_APP are executables which are not part of the normal boot processing but can be optionally chosen for execution if boot menu is provided, or via Hot Keys. See Section 3.1.6 for details.

Boot options with reserved category values, will be ignored by the boot manager.

If any Boot#### load option is marked as LOAD_OPTION_HIDDEN, then the load option will not appear in the menu (if any) provided by the boot manager for load option selection.

3.1.4 Boot Manager Capabilities

The boot manager can report its capabilities through the global variable BootOptionSupport. If the global variable is not present, then an installer or application must act as if a value of 0 was returned.

#define EFI_BOOT_OPTION_SUPPORT_KEY 0x00000001
#define EFI_BOOT_OPTION_SUPPORT_APP 0x00000002
#define EFI_BOOT_OPTION_SUPPORT_SYSPREP 0x00000010
#define EFI_BOOT_OPTION_SUPPORT_COUNT 0x00000300

If EFI_BOOT_OPTION_SUPPORT_KEY is set then the boot manager supports launching of Boot#### load options using key presses. If EFI_BOOT_OPTION_SUPPORT_APP is set then the boot manager supports boot options with LOAD_OPTION_CATEGORY_APP. If EFI_BOOT_OPTION_SUPPORT_SYSPREP is set then the boot manager supports boot options of form SysPrep####.

The value specified in EFI_BOOT_OPTION_SUPPORT_COUNT describes the maximum number of key presses which the boot manager supports in the EFI_KEY_OPTION.KeyData.InputKeyCount. This value is only valid if EFI_BOOT_OPTION_SUPPORT_KEY is set. Key sequences with more keys specified are ignored.

3.1.5 Launching Boot#### Applications

The boot manager may support a separate category of Boot#### load option for applications. The boot manager indicates that it supports this separate category by setting the EFI_BOOT_OPTION_SUPPORT_APP in the BootOptionSupport global variable.
When an application’s Boot#### option is being added to the BootOrder, the installer should clear LOAD_OPTION_ACTIVE so that the boot manager does not attempt to automatically “boot” the application. If the boot manager indicates that it supports a separate application category, as described above, the installer should set LOAD_OPTION_CATEGORY_APP. If not, it should set LOAD_OPTION_CATEGORY_BOOT.

3.1.6 Launching Boot#### Load Options Using Hot Keys

The boot manager may support launching a Boot#### load option using a special key press. If so, the boot manager reports this capability by setting EFI_BOOT_OPTION_SUPPORT_KEY in the BootOptionSupport global variable.

A boot manager which supports key press launch reads the current key information from the console. Then, if there was a key press, it compares the key returned against zero or more Key#### global variables. If it finds a match, it verifies that the Boot#### load option specified is valid and, if so, attempts to launch it immediately. The #### in the Key#### is a printable hexadecimal number (‘0’-‘9’, ‘A’-‘F’) with leading zeroes. The order which the Key#### variables are checked is implementation-specific.

The boot manager may ignore Key#### variables where the hot keys specified overlap with those used for internal boot manager functions. It is recommended that the boot manager delete these keys.

The Key#### variables have the following format:

Prototype

typedef struct _EFI_KEY_OPTION {
    EFI_BOOT_KEY_DATA KeyData;
    UINT32 BootOptionCrc;
    UINT16 BootOption;
    // EFI_INPUT_KEY Keys[];
} EFI_KEY_OPTION;

Parameters

KeyData

Specifies options about how the key will be processed. Type EFI_BOOT_KEY_DATA is defined in “Related Definitions” below.

BootOptionCrc

The CRC-32 which should match the CRC-32 of the entire EFI_LOAD_OPTION to which BootOption refers. If the CRC-32s do not match this value, then this key option is ignored.

BootOption

The Boot#### option which will be invoked if this key is pressed and the boot option is active (LOAD_OPTION_ACTIVE is set).

Keys

The key codes to compare against those returned by the EFI_SIMPLE_TEXT_INPUT and EFI_SIMPLE_TEXT_INPUT_EX protocols. The
number of key codes (0-3) is specified by the **EFI_KEY_CODE_COUNT** field in KeyOptions.

**Related Definitions**

```c
typedef union {
    struct {
        UINT32 Revision : 8;
        UINT32 ShiftPressed : 1;
        UINT32 ControlPressed : 1;
        UINT32 AltPressed : 1;
        UINT32 LogoPressed : 1;
        UINT32 MenuPressed : 1;
        UINT32 SysReqPressed : 1;
        UINT32 Reserved : 16;
        UINT32 InputKeyCount : 2;
    } Options;
    UINT32 PackedValue;
} EFI_BOOT_KEY_DATA;
```

- **Revision**
  
  Indicates the revision of the **EFI_KEY_OPTION** structure. This revision level should be 0.

- **ShiftPressed**
  
  Either the left or right Shift keys must be pressed (1) or must not be pressed (0).

- **ControlPressed**
  
  Either the left or right Control keys must be pressed (1) or must not be pressed (0).

- **AltPressed**
  
  Either the left or right Alt keys must be pressed (1) or must not be pressed (0).

- **LogoPressed**
  
  Either the left or right Logo keys must be pressed (1) or must not be pressed (0).

- **MenuPressed**
  
  The Menu key must be pressed (1) or must not be pressed (0).

- **SysReqPressed**
  
  The SysReq key must be pressed (1) or must not be pressed (0).

- **InputKeyCount**
  
  Specifies the actual number of entries in **EFI_KEY_OPTION.Keys**, from 0-3. If zero, then only the shift state is considered. If more than one, then the boot option will only be launched if all of the specified keys are pressed with the same shift state.

Example #1: ALT is the hot key. `KeyData.PackedValue = 0x00000400`.

Example #2: CTRL-ALT-P-R. `KeyData.PackedValue = 0x80000600`.
3.1.7 Required System Preparation Applications

A load option of the form `SysPrep####` is intended to designate a UEFI application that is required to execute in order to complete system preparation prior to processing of any `Boot####` variables. The execution order of `SysPrep####` applications is determined by the contents of the variable `SysPrepOrder` in a way directly analogous to the ordering of `Boot####` options by `BootOrder`.

The platform is required to examine all `SysPrep####` variables referenced in `SysPrepOrder`. If Attributes bit `LOAD_OPTION_ACTIVE` is set, and the application referenced by `FilePathList[0]` is present, the UEFI Applications thus identified must be loaded and launched in the order they appear in `SysPrepOrder` and prior to the launch of any load options of type `Boot####`.

When launched, the platform is required to provide the application loaded by `SysPrep####`, with the same services such as console and network as are normally provided at launch to applications referenced by a `Boot####` variable. `SysPrep####` application must exit and may not call `ExitBootServices()`. Processing of any Error Code returned at exit is according to system policy and does not necessarily change processing of following boot options. Any driver portion of the feature supported by `SysPrep####` boot option that is required to remain resident should be loaded by use of `Driver####` variable.

The Attributes option `LOAD_OPTION_FORCE_RECONNECT` is ignored for `SysPrep####` variables, and in the event that an application so launched performs some action that adds to the available hardware or drivers, the system preparation application shall itself utilize appropriate calls to `ConnectController()` or `DisconnectController()` to revise connections between drivers and hardware.

After all `SysPrep####` variables have been launched and exited, the platform shall notify `EFI_EVENT_GROUP_READY_TO_BOOT` and `EFI_EVENT_GROUP_AFTER_READY_TO_BOOT` event groups and begin to evaluate `Boot####` variables with Attributes set to `LOAD_OPTION_CATEGORY_BOOT` according to the order defined by `BootOrder`. The `FilePathList` of variables marked `LOAD_OPTION_CATEGORY_BOOT` shall not be evaluated prior to the completion of `EFI_EVENT_GROUP_AFTER_READY_TO_BOOT` event group processing.

3.2 Boot Manager Policy Protocol

**EFI_BOOT_MANAGER_POLICY_PROTOCOL**

Summary

This protocol is used by EFI Applications to request the UEFI Boot Manager to connect devices using platform policy.
GUID

```c
#define EFI_BOOT_MANAGER_POLICY_PROTOCOL_GUID \ 
  { 0xFEDF8E0C, 0xE147, 0x11E3,\ 
    { 0x99, 0x03, 0xB8, 0xE8, 0x56, 0x2C, 0xBA, 0xFA } }
```

Protocol Interface Structure

```c
typedef struct _EFI_BOOT_MANAGER_POLICY_PROTOCOL
  EFI_BOOT_MANAGER_POLICY_PROTOCOL;
struct _EFI_BOOT_MANAGER_POLICY_PROTOCOL {
  UINT64 Revision;
  EFI_BOOT_MANAGER_POLICY_CONNECT_DEVICE_PATH ConnectDevicePath;
  EFI_BOOT_MANAGER_POLICY_CONNECT_DEVICE_CLASS ConnectDeviceClass;
};
```

- **ConnectDevicePath**: Connect a Device Path following the platforms EFI Boot Manager policy.
- **ConnectDeviceClass**: Connect a class of devices, named by `EFI_GUID`, following the platforms UEFI Boot Manager policy.

Description

The `EFI_BOOT_MANAGER_POLICY_PROTOCOL` is produced by the platform firmware to expose Boot Manager policy and platform specific `EFI_BOOT_SERVICES.ConnectController()` behavior.

Related Definitions

```c
#define EFI_BOOT_MANAGER_POLICY_PROTOCOL_REVISION 0x00010000
```

`EFI_BOOT_MANAGER_POLICY_PROTOCOL.ConnectDevicePath()`

Summary

Connect a device path following the platform’s EFI Boot Manager policy.

Prototype

```c
typedef
  EFI_STATUS
(EFIAPI *EFI_BOOT_MANAGER_POLICY_CONNECT_DEVICE_PATH)(
  IN EFI_BOOT_MANAGER_POLICY_PROTOCOL *This,
  IN EFI_DEVICE_PATH *DevicePath,
  IN BOOLEAN Recursive)
);
```

Parameters

- **This**: A pointer to the `EFI_BOOT_MANAGER_POLICY_PROTOCOL` instance. Type `EFI_BOOT_MANAGER_POLICY_PROTOCOL` defined above.
- **DevicePath**: Points to the start of the EFI device path to connect. If `DevicePath` is NULL then all the controllers in the system will be connected using the platform’s EFI Boot Manager policy.
Recursive

If **TRUE**, then `ConnectController()` is called recursively until the entire tree of controllers below the controller specified by `DevicePath` have been created. If **FALSE**, then the tree of controllers is only expanded one level. If `DevicePath` is NULL then `Recursive` is ignored.

**Description**

The `ConnectDevicePath()` function allows the caller to connect a `DevicePath` using the same policy as the EFI Boot Manager.

If `Recursive` is **TRUE**, then `ConnectController()` is called recursively until the entire tree of controllers below the controller specified by `DevicePath` have been created. If `Recursive` is **FALSE**, then the tree of controllers is only expanded one level. If `DevicePath` is NULL then `Recursive` is ignored.

**Status Codes Returned**

<table>
<thead>
<tr>
<th>Code</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>EFI_SUCCESS</td>
<td>The <code>DevicePath</code> was connected</td>
</tr>
<tr>
<td>EFI_NOT_FOUND</td>
<td>The <code>DevicePath</code> was not found</td>
</tr>
<tr>
<td>EFI_NOT_FOUND</td>
<td>No driver was connected to <code>DevicePath</code>.</td>
</tr>
<tr>
<td>EFI_SECURITY_VIOLATION</td>
<td>The user has no permission to start UEFI device drivers on the <code>DevicePath</code>.</td>
</tr>
<tr>
<td>EFI_UNSUPPORTED</td>
<td>The current TPL is not <code>TPL_APPLICATION</code>.</td>
</tr>
</tbody>
</table>

**EFI_BOOT_MANAGER_POLICY_PROTOCOL.ConnectDeviceClass()**

**Summary**

Connect a class of devices using the platform Boot Manager policy.

**Prototype**

```c
typedef
  EFI_STATUS
  (EFIAPI *EFI_BOOT_MANAGER_POLICY_CONNECT_DEVICE_CLASS)(
    IN EFI_BOOT_MANAGER_POLICY_PROTOCOL  *This,
    IN EFI_GUID                          *Class
  );
```

**Parameters**

- **This**
  A pointer to the `EFI_BOOT_MANAGER_POLICY_PROTOCOL` instance. Type `EFI_BOOT_MANAGER_POLICY_PROTOCOL` is defined above.

- **Class**
  A pointer to an `EFI_GUID` that represents a class of devices that will be connected using the Boot Manager’s platform policy.

**Description**

The `ConnectDeviceClass()` function allows the caller to request that the Boot Manager connect a class of devices.
If \texttt{Class} is \texttt{EFI\_BOOT\_MANAGER\_POLICY\_CONSOLE\_GUID} then the Boot Manager will use platform policy to connect consoles. Some platforms may restrict the number of consoles connected as they attempt to fast boot, and calling \texttt{ConnectDeviceClass()} with a \texttt{Class} value of \texttt{EFI\_BOOT\_MANAGER\_POLICY\_CONSOLE\_GUID} must connect the set of consoles that follow the Boot Manager platform policy, and the \texttt{EFI\_SIMPLE\_TEXT\_INPUT\_PROTOCOL}, \texttt{EFI\_SIMPLE\_TEXT\_INPUT\_EX\_PROTOCOL}, and the \texttt{EFI\_SIMPLE\_TEXT\_OUTPUT\_PROTOCOL} are produced on the connected handles. The Boot Manager may restrict which consoles get connect due to platform policy, for example a security policy may require that a given console is not connected.

If \texttt{Class} is \texttt{EFI\_BOOT\_MANAGER\_POLICY\_NETWORK\_GUID} then the Boot Manager will connect the protocols the platform supports for UEFI general purpose network applications on one or more handles. The protocols associated with UEFI general purpose network applications are defined in Section 2.6.2, list item number 7. If more than one network controller is available a platform will connect, one, many, or all of the networks based on platform policy. Connecting UEFI networking protocols, like \texttt{EFI\_DHCP4\_PROTOCOL}, does not establish connections on the network. The UEFI general purpose network application that called \texttt{ConnectDeviceClass()} may need to use the published protocols to establish the network connection. The Boot Manager can optionally have a policy to establish a network connection.

If \texttt{Class} is \texttt{EFI\_BOOT\_MANAGER\_POLICY\_CONNECT\_ALL\_GUID} then the Boot Manager will connect all UEFI drivers using the UEFI Boot Service \texttt{EFI\_BOOT\_SERVICES\_ConnectController()}. If the Boot Manager has policy associated with connect all UEFI drivers this policy will be used.

A platform can also define platform specific \texttt{Class} values as a properly generated \texttt{EFI\_GUID} would never conflict with this specification.

### Related Definitions

```c
#define EFI_BOOT_MANAGER_POLICY_CONSOLE_GUID \ { 0xCAB0E94C, 0xE15F, 0x11E3,\ { 0x91, 0x8D, 0xBB, 0xE8, 0x56, 0x2C, 0xBA, 0xFA } } 
#define EFI_BOOT_MANAGER_POLICY_NETWORK_GUID \ { 0xD04159DC, 0xE15F, 0x11E3,\ { 0xB2, 0x61, 0xBB, 0xE8, 0x56, 0x2C, 0xBA, 0xFA } } 
#define EFI_BOOT_MANAGER_POLICY_CONNECT_ALL_GUID \ { 0x113B2126, 0xFC8A, 0x11E3,\ { 0xBD, 0x6C, 0xBB, 0xE8, 0x56, 0x2C, 0xBA, 0xFA } } 
```

### Status Codes Returned

<table>
<thead>
<tr>
<th>Status Code</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>EFI_SUCCESS</td>
<td>At least one devices of the \texttt{Class} was connected.</td>
</tr>
<tr>
<td>EFI_DEVICE_ERROR</td>
<td>Devices were not connected due to an error.</td>
</tr>
<tr>
<td>EFI_NOT_FOUND</td>
<td>The \texttt{Class} is not supported by the platform.</td>
</tr>
<tr>
<td>EFI_UNSUPPORTED</td>
<td>The current TPL is not \texttt{TPL_APPLICATION}.</td>
</tr>
</tbody>
</table>

### 3.3 Globally Defined Variables

This section defines a set of variables that have architecturally defined meanings. In addition to the defined data content, each such variable has an architecturally defined attribute that indicates when the
data variable may be accessed. The variables with an attribute of NV are nonvolatile. This means that their values are persistent across resets and power cycles. The value of any environment variable that does not have this attribute will be lost when power is removed from the system and the state of firmware reserved memory is not otherwise preserved. The variables with an attribute of BS are only available before EFI_BOOT_SERVICES. ExitBootServices() is called. This means that these environment variables can only be retrieved or modified in the preboot environment. They are not visible to an operating system. Environment variables with an attribute of RT are available before and after ExitBootServices() is called. Environment variables of this type can be retrieved and modified in the preboot environment, and from an operating system. The variables with an attribute of AT are variables with a time-based authenticated write access defined in Section 8.2.1. All architecturally defined variables use the EFI_GLOBAL_VARIABLE VendorGuid:

```
#define EFI_GLOBAL_VARIABLE \
{0x8BE4DF61,0x93CA,0x11d2,\ 
 {0xAA,0x0D,0x00,0xE0,0x98,0x03,0x2B,0x8C}}
```

To prevent name collisions with possible future globally defined variables, other internal firmware data variables that are not defined here must be saved with a unique VendorGuid other than EFI_GLOBAL_VARIABLE or any other GUID defined by the UEFI Specification. Implementations must only permit the creation of variables with a UEFI Specification-defined VendorGuid when these variables are documented in the UEFI Specification.

### Table 3-1 Global Variables

<table>
<thead>
<tr>
<th>Variable Name</th>
<th>Attribute</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>AuditMode</td>
<td>BS, RT</td>
<td>Whether the system is operating in Audit Mode (1) or not (0). All other values are reserved. Should be treated as read-only except when DeployedMode is 0. Always becomes read-only after ExitBootServices() is called.</td>
</tr>
<tr>
<td>Boot####</td>
<td>NV, BS, RT</td>
<td>A boot load option. #### is a printed hex value. No 0x or h is included in the hex value.</td>
</tr>
<tr>
<td>BootCurrent</td>
<td>BS, RT</td>
<td>The boot option that was selected for the current boot.</td>
</tr>
<tr>
<td>BootNext</td>
<td>NV, BS, RT</td>
<td>The boot option for the next boot only.</td>
</tr>
<tr>
<td>BootOrder</td>
<td>NV, BS, RT</td>
<td>The ordered boot option load list.</td>
</tr>
<tr>
<td>BootOptionSupport</td>
<td>BS,RT,</td>
<td>The types of boot options supported by the boot manager. Should be treated as read-only.</td>
</tr>
<tr>
<td>ConIn</td>
<td>NV, BS, RT</td>
<td>The device path of the default input console.</td>
</tr>
<tr>
<td>ConInDev</td>
<td>BS, RT</td>
<td>The device path of all possible console input devices.</td>
</tr>
<tr>
<td>ConOut</td>
<td>NV, BS, RT</td>
<td>The device path of the default output console.</td>
</tr>
<tr>
<td>ConOutDev</td>
<td>BS, RT</td>
<td>The device path of all possible console output devices.</td>
</tr>
<tr>
<td>dBDefault</td>
<td>BS, RT</td>
<td>The OEM's default secure boot signature store. Should be treated as read-only.</td>
</tr>
<tr>
<td>dbrDefault</td>
<td>BS, RT</td>
<td>The OEM's default OS Recovery signature store. Should be treated as read-only.</td>
</tr>
<tr>
<td>dbtDefault</td>
<td>BS, RT</td>
<td>The OEM's default secure boot timestamp signature store. Should be treated as read-only.</td>
</tr>
<tr>
<td>dbxDefault</td>
<td>BS, RT</td>
<td>The OEM's default secure boot blacklist signature store. Should be treated as read-only.</td>
</tr>
<tr>
<td>Variable Name</td>
<td>Attribute</td>
<td>Description</td>
</tr>
<tr>
<td>--------------------</td>
<td>-----------</td>
<td>---------------------------------------------------------------------------------------------------------------------------------------------</td>
</tr>
<tr>
<td>DeployedMode</td>
<td>BS, RT</td>
<td>Whether the system is operating in Deployed Mode (1) or not (0). All other values are reserved. Should be treated as read-only when its value is 1. Always becomes read-only after ExitBootServices() is called.</td>
</tr>
<tr>
<td>Driver####</td>
<td>NV, BS, RT</td>
<td>A driver load option. #### is a printed hex value.</td>
</tr>
<tr>
<td>DriverOrder</td>
<td>NV, BS, RT</td>
<td>The ordered driver load option list.</td>
</tr>
<tr>
<td>ErrOut</td>
<td>NV, BS, RT</td>
<td>The device path of the default error output device.</td>
</tr>
<tr>
<td>ErrOutDev</td>
<td>BS, RT</td>
<td>The device path of all possible error output devices.</td>
</tr>
<tr>
<td>HwErrRecSupport</td>
<td>NV, BS, RT</td>
<td>Identifies the level of hardware error record persistence support implemented by the platform. This variable is only modified by firmware and is read-only to the OS.</td>
</tr>
<tr>
<td>KEK</td>
<td>NV, BS, RT, AT</td>
<td>The Key Exchange Key Signature Database.</td>
</tr>
<tr>
<td>KEKDefault</td>
<td>BS, RT</td>
<td>The OEM's default Key Exchange Key Signature Database. Should be treated as read-only.</td>
</tr>
<tr>
<td>Key####</td>
<td>NV, BS, RT</td>
<td>Describes hot key relationship with a Boot#### load option.</td>
</tr>
<tr>
<td>Lang</td>
<td>NV, BS, RT</td>
<td>The language code that the system is configured for. This value is deprecated.</td>
</tr>
<tr>
<td>LangCodes</td>
<td>BS, RT</td>
<td>The language codes that the firmware supports. This value is deprecated.</td>
</tr>
<tr>
<td>OsIndications</td>
<td>NV, BS, RT</td>
<td>Allows the OS to request the firmware to enable certain features and to take certain actions.</td>
</tr>
<tr>
<td>OsIndicationsSupported</td>
<td>BS, RT</td>
<td>Allows the firmware to indicate supported features and actions to the OS.</td>
</tr>
<tr>
<td>OsRecoveryOrder</td>
<td>BS, RT, NV, AT</td>
<td>OS-specified recovery options.</td>
</tr>
<tr>
<td>PK</td>
<td>NV, BS, RT, AT</td>
<td>The public Platform Key.</td>
</tr>
<tr>
<td>PKDefault</td>
<td>BS, RT</td>
<td>The OEM's default public Platform Key. Should be treated as read-only.</td>
</tr>
<tr>
<td>PlatformLangCodes</td>
<td>BS, RT</td>
<td>The language codes that the firmware supports.</td>
</tr>
<tr>
<td>PlatformLang</td>
<td>NV, BS, RT</td>
<td>The language code that the system is configured for.</td>
</tr>
<tr>
<td>PlatformRecovery####</td>
<td>BS, RT</td>
<td>Platform-specified recovery options. These variables are only modified by firmware and are read-only to the OS.</td>
</tr>
<tr>
<td>SignatureSupport</td>
<td>BS, RT</td>
<td>Array of GUIDs representing the type of signatures supported by the platform firmware. Should be treated as read-only.</td>
</tr>
<tr>
<td>SecureBoot</td>
<td>BS, RT</td>
<td>Whether the platform firmware is operating in Secure boot mode (1) or not (0). All other values are reserved. Should be treated as read-only.</td>
</tr>
<tr>
<td>SetupMode</td>
<td>BS, RT</td>
<td>Whether the system should require authentication on SetVariable() requests to Secure Boot policy variables (0) or not (1). Should be treated as read-only. The system is in &quot;Setup Mode&quot; when SetupMode==1, AuditMode==0, and DeployedMode==0.</td>
</tr>
<tr>
<td>SysPrep####</td>
<td>NV, BS, RT</td>
<td>A System Prep application load option containing a EFI_LOAD_OPTION descriptor. #### is a printed hex value.</td>
</tr>
<tr>
<td>SysPrepPrepOrder</td>
<td>NV, BS, RT</td>
<td>The ordered System Prep Application load option list.</td>
</tr>
</tbody>
</table>
The `PlatformLangCodes` variable contains a null-terminated ASCII string representing the language codes that the firmware can support. At initialization time the firmware computes the supported languages and creates this data variable. Since the firmware creates this value on each initialization, its contents are not stored in nonvolatile memory. This value is considered read-only. `PlatformLangCodes` is specified in Native RFC 4646 format. See Appendix M. `LangCodes` is deprecated and may be provided for backwards compatibility.

The `PlatformLang` variable contains a null-terminated ASCII string language code that the machine has been configured for. This value may be changed to any value supported by `PlatformLangCodes`. If this change is made in the preboot environment, then the change will take effect immediately. If this change is made at OS runtime, then the change does not take effect until the next boot. If the language code is set to an unsupported value, the firmware will choose a supported default at initialization and set `PlatformLang` to a supported value. `PlatformLang` is specified in Native RFC 4646 array format. See Appendix M. `Lang` is deprecated and may be provided for backwards compatibility.

`Lang` has been deprecated. If the platform supports this variable, it must map any changes in the `Lang` variable into `PlatformLang` in the appropriate format.

`Langcodes` has been deprecated. If the platform supports this variable, it must map any changes in the `Langcodes` variable into `PlatformLang` in the appropriate format.

The `Timeout` variable contains a binary `UINT16` that supplies the number of seconds that the firmware will wait before initiating the original default boot selection. A value of 0 indicates that the default boot selection is to be initiated immediately on boot. If the value is not present, or contains the value of 0xFFFF then firmware will wait for user input before booting. This means the default boot selection is not automatically started by the firmware.

The `ConIn`, `ConOut`, and `ErrOut` variables each contain an `EFI_DEVICE_PATH_PROTOCOL` descriptor that defines the default device to use on boot. Changes to these values made in the preboot environment take effect immediately. Changes to these values at OS runtime do not take effect until the next boot. If the firmware cannot resolve the device path, it is allowed to automatically replace the values, as needed, to provide a console for the system. If the device path starts with a USB Class device path (see Table 10-25), then any input or output device that matches the device path must be used as a console if it is supported by the firmware.

The `ConInDev`, `ConOutDev`, and `ErrOutDev` variables each contain an `EFI_DEVICE_PATH_PROTOCOL` descriptor that defines all the possible default devices to use on boot. These variables are volatile, and are set dynamically on every boot. `ConIn`, `ConOut`, and `ErrOut` are always proper subsets of `ConInDev`, `ConOutDev`, and `ErrOutDev`.

Each `Boot####` variable contains an `EFI_LOAD_OPTION`. Each `Boot####` variable is the name “Boot” appended with a unique four digit hexadecimal number. For example, `Boot0001`, `Boot0002`, `Boot0A02`, etc.
The **OsRecoveryOrder** variable contains an array of **EFI_GUID** structures. Each **EFI_GUID** structure specifies a namespace for variables containing OS-defined recovery entries (see Section 3.4.1). Write access to this variable is controlled by the security key database dbr (see Section 8.2.1).

PlatformRecovery variables share the same structure as Boot variables. These variables are processed when the system is performing recovery of boot options.

The **BootOrder** variable contains an array of **UINT16**’s that make up an ordered list of the Boot options. The first element in the array is the value for the first logical boot option, the second element is the value for the second logical boot option, etc. The **BootOrder** order list is used by the firmware’s boot manager as the default boot order.

The **BootNext** variable is a single **UINT16** that defines the Boot option that is to be tried first on the next boot. After the **BootNext** boot option is tried the normal **BootOrder** list is used. To prevent loops, the boot manager deletes this variable before transferring control to the preselected boot option.

The **BootCurrent** variable is a single **UINT16** that defines the Boot option that was selected on the current boot.

The **BootOptionSupport** variable is a **UINT32** that defines the types of boot options supported by the boot manager.

Each Driver variable contains an **EFI_LOAD_OPTION**. Each load option variable is appended with a unique number, for example Driver0001, Driver0002, etc.

The **DriverOrder** variable contains an array of **UINT16**’s that make up an ordered list of the Driver variable. The first element in the array is the value for the first logical driver load option, the second element is the value for the second logical driver load option, etc. The **DriverOrder** list is used by the firmware’s boot manager as the default load order for UEFI drivers that it should explicitly load.

The **Key** variable associates a key press with a single boot option. Each **Key** variable is the name "Key" appended with a unique four digit hexadecimal number. For example, Key0001, Key0002, Key00A0, etc.

The **HwErrRecSupport** variable contains a binary UINT16 that supplies the level of support for Hardware Error Record Persistence (see Section 8.2.4) that is implemented by the platform. If the value is not present, then the platform implements no support for Hardware Error Record Persistence. A value of zero indicates that the platform implements no support for Hardware Error Record Persistence. A value of 1 indicates that the platform implements Hardware Error Record Persistence as defined in Section 8.2.4. Firmware initializes this variable. All other values are reserved for future use.

The **SetupMode** variable is an 8-bit unsigned integer that defines whether the system should require authentication (0) or not (1) on **SetVariable()** requests to Secure Boot Policy Variables. Secure Boot Policy Variables include:

- The global variables PK, KEK, and **OsRecoveryOrder**
- All variables named **OsRecovery** under all VendorGuids
- All variables with the VendorGuid **EFI_IMAGE_SECURITY_DATABASE_GUID**.

Secure Boot Policy Variables must be created using the **EFI_VARIABLE_AUTHENTICATION_2** structure.
The **AuditMode** variable is an 8-bit unsigned integer that defines whether the system is currently operating in Audit Mode.

The **DeployedMode** variable is an 8-bit unsigned integer that defines whether the system is currently operating in Deployed Mode.

The **KEK** variable contains the current Key Exchange Key database.

The **PK** variable contains the current Platform Key.

The **VendorKeys** variable is an 8-bit unsigned integer that defines whether the Security Boot Policy Variables have been modified by anyone other than the platform vendor or a holder of the vendor-provided keys. A value of 0 indicates that someone other than the platform vendor or a holder of the vendor-provided keys has modified the Secure Boot Policy Variables Otherwise, the value will be 1.

The **KEKDefault** variable, if present, contains the platform-defined Key Exchange Key database. This is not used at runtime but is provided in order to allow the OS to recover the OEM's default key setup. The contents of this variable do not include an **EFI_VARIABLE_AUTHENTICATION** or **EFI_VARIABLE_AUTHENTICATION2** structure.

The **PKDefault** variable, if present, contains the platform-defined Platform Key. This is not used at runtime but is provided in order to allow the OS to recover the OEM's default key setup. The contents of this variable do not include an **EFI_VARIABLE_AUTHENTICATION2** structure.

The **dbDefault** variable, if present, contains the platform-defined secure boot signature database. This is not used at runtime but is provided in order to allow the OS to recover the OEM's default key setup. The contents of this variable do not include an **EFI_VARIABLE_AUTHENTICATION2** structure.

The **dbrDefault** variable, if present, contains the platform-defined secure boot authorized recovery signature database. This is not used at runtime but is provided in order to allow the OS to recover the OEM's default key setup. The contents of this variable do not include an **EFI_VARIABLE_AUTHENTICATION2** structure.

The **dbtDefault** variable, if present, contains the platform-defined secure boot timestamp signature database. This is not used at runtime but is provided in order to allow the OS to recover the OEM's default key setup. The contents of this variable do not include an **EFI_VARIABLE_AUTHENTICATION2** structure.

The **dbxDefault** variable, if present, contains the platform-defined secure boot blacklist signature database. This is not used at runtime but is provided in order to allow the OS to recover the OEM's default key setup. The contents of this variable do not include an **EFI_VARIABLE_AUTHENTICATION2** structure.

The **SignatureSupport** variable returns an array of GUIDs, with each GUID representing a type of signature which the platform firmware supports for images and other data. The different signature types are described in "Signature Database".

The **SecureBoot** variable is an 8-bit unsigned integer that defines whether the platform firmware is operating with Secure Boot enabled. A value of 1 indicates that platform firmware performs driver and boot application signature verification as specified in **UEFI Image Validation** during the current boot. A value of 0 indicates that driver and boot application signature verification is not active during the current boot. The SecureBoot variable is initialized prior to Secure Boot image authentication and thereafter
should be treated as read-only and immutable. Its initialization value is determined by platform policy but must be 0 if the platform is in Setup Mode or Audit Mode during its initialization.

The `OsIndicationsSupported` variable indicates which of the OS indication features and actions that the firmware supports. This variable is recreated by firmware every boot, and cannot be modified by the OS (see `SetVariable()` Attributes usage rules once `ExitBootServices()` is performed).

The `OsIndications` variable is used to indicate which features the OS wants firmware to enable or which actions the OS wants the firmware to take. The OS will supply this data with a `SetVariable()` call. See Section 8.5.4 for the variable definition.

### 3.4 Boot Option Recovery

Boot option recovery consists of two independent parts, operating system-defined recovery and platform-defined recovery. OS-defined recovery is an attempt to allow installed operating systems to recover any needed boot options, or to launch full operating system recovery. Platform-defined recovery includes any remedial actions performed by the platform as a last resort when no operating system is found, such as the `Default Boot Behavior` (see Section 3.4.3). This could include behaviors such as warranty service reconfiguration or diagnostic options.

In the event that boot option recovery must be performed, the boot manager must first attempt OS-defined recovery, re-attempt normal booting via `Boot###` and `BootOrder` variables, and finally attempt platform-defined recovery if no options have succeeded.

#### 3.4.1 OS-Defined Boot Option Recovery

If the `EFI_OS_INDICATIONS_START_OS_RECOVERY` bit is set in `OsIndications`, or if processing of `BootOrder` does not result in success, the platform must process OS-defined recovery options. In the case where OS-defined recovery is entered due to `OsIndications`, `SysPrepOrder` and `SysPrep###` variables should not be processed. Note that in order to avoid ambiguity in intent, this bit is ignored in `OsIndications` if `EFI_OS_INDICATIONS_START_PLATFORM_RECOVERY` is set.

OS-defined recovery uses the `OsRecoveryOrder` variable, as well as variables created with vendor specific `VendorGuid` values and a name following the pattern `OsRecovery###`. Each of these variables must be an authenticated variable with the `EFI_VARIABLE_TIME_BASED_AUTHENTICATED_WRITE_ACCESS` attribute set.

To process these variables, the boot manager iterates over the array of `EFI_GUID` structures in the `OsRecoveryOrder` variable, and each `GUID` specified is treated as a `VendorGuid` associated with a series of variable names. For each `GUID`, the firmware attempts to load and execute, in hexadecimal sort order, every variable with that `GUID` and a name following the pattern `OsRecovery###`. These variables have the same format as `Boot###` variables, and the boot manager must verify that each variable it attempts to load was created with a public key that is associated with a certificate chaining to one listed in the authorized recovery signature database `dbr` and not in the forbidden signature database, or is created by a key in the Key Exchange Key database `KEK` or the current Platform Key `PK`.

If the boot manager finishes processing `OsRecovery###` options without `EFI_BOOT SERVICES.ExitBootServices()` or `ResetSystem()` having been called, it must attempt to process `BootOrder` a second time. If booting does not succeed during that process, OS-defined recovery has failed, and the boot manager must attempt platform-based recovery.
If, while processing `OsRecovery` variables, the boot manager encounters an entry which cannot be loaded or executed due to a security policy violation, it must ignore that variable.

### 3.4.2 Platform-Defined Boot Option Recovery

If the `EFI_OS_INDICATIONS_START_PLATFORM_RECOVERY` bit is set in `OsIndications`, or if OS-defined recovery has failed, the system firmware must commence with platform-specific recovery by iterating its `PlatformRecovery` variables in the same manner as `OsRecovery`, but must stop processing if any entry is successful. In the case where platform-specific recovery is entered due to `OsIndications`, `SysPrepOrder` and `SysPrep` variables should not be processed.

### 3.4.3 Boot Option Variables Default Boot Behavior

The default state of globally-defined variables is firmware vendor specific. However, the boot options require a standard default behavior in the exceptional case that valid boot options are not present on a platform. The default behavior must be invoked any time the `BootOrder` variable does not exist or only points to nonexistent boot options, or if no entry in `BootOrder` can successfully be executed.

If system firmware supports boot option recovery as described in Section 3.4, system firmware must include a `PlatformRecovery` variable specifying a short-form File Path Media Device Path (see Section 3.1.2) containing the platform default file path for removable media (see Table 3-2). It is recommended for maximal compatibility with prior versions of this specification that this entry be the first such variable, though it may be at any position within the list.

It is expected that this default boot will load an operating system or a maintenance utility. If this is an operating system setup program, it is then responsible for setting the requisite environment variables for subsequent boots. The platform firmware may also decide to recover or set to a known set of boot options.

### 3.5 Boot Mechanisms

EFI can boot from a device using the `EFI_SIMPLE_FILE_SYSTEM_PROTOCOL` or the `EFI_LOAD_FILE_PROTOCOL`. A device that supports the `EFI_SIMPLE_FILE_SYSTEM_PROTOCOL` must materialize a file system protocol for that device to be bootable. If a device does not wish to support a complete file system, it may produce an `EFI_LOAD_FILE_PROTOCOL` which allows it to materialize an image directly. The Boot Manager will attempt to boot using the `EFI_SIMPLE_FILE_SYSTEM_PROTOCOL` first. If that fails, then the `EFI_LOAD_FILE_PROTOCOL` will be used.

#### 3.5.1 Boot via the Simple File Protocol

When booting via the `EFI_SIMPLE_FILE_SYSTEM_PROTOCOL`, the `FilePath` will start with a device path that points to the device that implements the `EFI_SIMPLE_FILE_SYSTEM_PROTOCOL` or the `EFI_BLOCK_IO_PROTOCOL`. The next part of the `FilePath` may point to the file name, including subdirectories, which contain the bootable image. If the file name is a null device path, the file name must be generated from the rules defined below.

If the `FilePathList[0]` device does not support the `EFI_SIMPLE_FILE_SYSTEM_PROTOCOL`, but supports the `EFI_BLOCK_IO_PROTOCOL` protocol, then the EFI Boot Service `EFI_BOOT_SERVICES.ConnectController()` must be called for `FilePathList[0]` with
DriverImageHandle and RemainingDevicePath set to NULL and the Recursive flag is set to TRUE. The firmware will then attempt to boot from any child handles produced using the algorithms outlined below.

The format of the file system specified is contained in Section 13.3. While the firmware must produce an EFI_SIMPLE_FILE_SYSTEM_PROTOCOL that understands the UEFI file system, any file system can be abstracted with the EFI_SIMPLE_FILE_SYSTEM_PROTOCOL interface.

### 3.5.1.1 Removable Media Boot Behavior

To generate a file name when none is present in the FilePath, the firmware must append a default file name in the form \EFI\BOOT\BOOT{machine type short-name}.EFI where machine type short-name defines a PE32+ image format architecture. Each file only contains one UEFI image type, and a system may support booting from one or more images types. Table 3-2 lists the UEFI image types.

<table>
<thead>
<tr>
<th>Media Architecture</th>
<th>File Name Convention</th>
<th>PE Executable Machine Type *</th>
</tr>
</thead>
<tbody>
<tr>
<td>32-bit</td>
<td>BOOTIA32.EFI</td>
<td>0x14c</td>
</tr>
<tr>
<td>x64</td>
<td>BOOTx64.EFI</td>
<td>0x8664</td>
</tr>
<tr>
<td>Itanium architecture</td>
<td>BOOTIA64.EFI</td>
<td>0x200</td>
</tr>
<tr>
<td>AArch32 architecture</td>
<td>BOOTARM.EFI</td>
<td>0x01c2</td>
</tr>
<tr>
<td>AArch64 architecture</td>
<td>BOOTAA64.EFI</td>
<td>0xAA64</td>
</tr>
<tr>
<td>RISC-V 32-bit architecture</td>
<td>BOOTRISCV32.EFI</td>
<td>0x5032</td>
</tr>
<tr>
<td><strong>RISC-V 64-bit architecture</strong></td>
<td>BOOTRISCV64.EFI</td>
<td>0x5064</td>
</tr>
<tr>
<td><strong>RISC-V 128-bit architecture</strong></td>
<td>BOOTRISCV128.EFI</td>
<td>0x5128</td>
</tr>
</tbody>
</table>

Note: * The PE Executable machine type is contained in the machine field of the COFF file header as defined in the Microsoft Portable Executable and Common Object File Format Specification, Revision 6.0

Media may support multiple architectures by simply having a \EFI\BOOT\BOOT{machine type short-name}.EFI file of each possible machine type.

### 3.5.2 Boot via the Load File Protocol

When booting via the EFI_LOAD_FILE_PROTOCOL protocol, the FilePath is a device path that points to a device that “speaks” the EFI_LOAD_FILE_PROTOCOL. The image is loaded directly from the device that supports the EFI_LOAD_FILE_PROTOCOL. The remainder of the FilePath will contain information that is specific to the device. Firmware passes this device-specific data to the loaded image, but does not use it to load the image. If the remainder of the FilePath is a null device path it is the loaded image’s responsibility to implement a policy to find the correct boot device.

The EFI_LOAD_FILE_PROTOCOL is used for devices that do not directly support file systems. Network devices commonly boot in this model where the image is materialized without the need of a file system.
3.5.2.1 Network Booting

Network booting is described by the Preboot eXecution Environment (PXE) BIOS Support Specification that is part of the Wired for Management Baseline specification. PXE specifies UDP, DHCP, and TFTP network protocols that a booting platform can use to interact with an intelligent system load server. UEFI defines special interfaces that are used to implement PXE. These interfaces are contained in the EFI_PXE_BASE_CODE_PROTOCOL (see Section 24.3).

3.5.2.2 Future Boot Media

Since UEFI defines an abstraction between the platform and the OS and its loader it should be possible to add new types of boot media as technology evolves. The OS loader will not necessarily have to change to support new types of boot. The implementation of the UEFI platform services may change, but the interface will remain constant. The OS will require a driver to support the new type of boot media so that it can make the transition from UEFI boot services to OS control of the boot media.
4 - EFI System Table

This section describes the entry point to a UEFI image and the parameters that are passed to that entry point. There are three types of UEFI images that can be loaded and executed by firmware conforming to this specification. These are UEFI applications (see Section 2.1.2), UEFI boot service drivers (see Section 2.1.4), and UEFI runtime drivers (see Section 2.1.4). UEFI applications include UEFI OS loaders (see Section 2.1.3). There are no differences in the entry point for these three image types.

4.1 UEFI Image Entry Point

The most significant parameter that is passed to an image is a pointer to the System Table. This pointer is EFI_IMAGE_ENTRY_POINT (see definition immediately below), the main entry point for a UEFI Image. The System Table contains pointers to the active console devices, a pointer to the Boot Services Table, a pointer to the Runtime Services Table, and a pointer to the list of system configuration tables such as ACPI, SMBIOS, and the SAL System Table. This section describes the System Table in detail.

EFI_IMAGE_ENTRY_POINT

Summary

This is the main entry point for a UEFI Image. This entry point is the same for UEFI applications and UEFI drivers.

Prototype

typedef
EFI_STATUS
(EFIAPPI *EFI_IMAGE_ENTRY_POINT) (  
IN EFI_HANDLE ImageHandle,  
IN EFI_SYSTEM_TABLE *SystemTable  
);

Parameters

ImageHandle The firmware allocated handle for the UEFI image.
SystemTable A pointer to the EFI System Table.

Description

This function is the entry point to an EFI image. An EFI image is loaded and relocated in system memory by the EFI Boot Service EFI_BOOT_SERVICES.LoadImage(). An EFI image is invoked through the EFI Boot Service EFI_BOOT_SERVICES.StartImage().

The first argument is the image’s image handle. The second argument is a pointer to the image’s system table. The system table contains the standard output and input handles, plus pointers to the EFI_BOOT_SERVICES and EFI_RUNTIME_SERVICES tables. The service tables contain the entry points in the firmware for accessing the core EFI system functionality. The handles in the system table are used to obtain basic access to the console. In addition, the System Table contains pointers to other standard tables that a loaded image may use if the associated pointers are initialized to nonzero values. Examples of such tables are ACPI, SMBIOS, SAL System Table, etc.
The **ImageHandle** is a firmware-allocated handle that is used to identify the image on various functions. The handle also supports one or more protocols that the image can use. All images support the `EFI_LOADED_IMAGE_PROTOCOL` and the `EFI_LOADED_IMAGE_DEVICE_PATH_PROTOCOL` that returns the source location of the image, the memory location of the image, the load options for the image, etc. The exact `EFI_LOADED_IMAGE_PROTOCOL` and `EFI_LOADED_IMAGE_DEVICE_PATH_PROTOCOL` structures are defined in Section 9.

If the UEFI image is a UEFI application that is not a UEFI OS loader, then the application executes and either returns or calls the EFI Boot Services `EFI_BOOT_SERVICES.Exit()`. A UEFI application is always unloaded from memory when it exits, and its return status is returned to the component that started the UEFI application.

If the UEFI image is a UEFI OS Loader, then the UEFI OS Loader executes and either returns, calls the EFI Boot Service `Exit()`, or calls the EFI Boot Service `EFI_BOOT_SERVICES.ExitBootServices()`. If the EFI OS Loader returns or calls `Exit()`, then the load of the OS has failed, and the EFI OS Loader is unloaded from memory and control is returned to the component that attempted to boot the UEFI OS Loader. If `ExitBootServices()` is called, then the UEFI OS Loader has taken control of the platform, and EFI will not regain control of the system until the platform is reset. One method of resetting the platform is through the EFI Runtime Service `ResetSystem()`.

If the UEFI image is a UEFI Driver, then the UEFI driver executes and either returns or calls the Boot Service `Exit()`. If the UEFI driver returns an error, then the driver is unloaded from memory. If the UEFI driver returns `EFI_SUCCESS`, then it stays resident in memory. If the UEFI driver does not follow the UEFI Driver Model, then it performs any required initialization and installs its protocol services before returning. If the driver does follow the UEFI Driver Model, then the entry point is not allowed to touch any device hardware. Instead, the entry point is required to create and install the `EFI_DRIVER_BINDING_PROTOCOL` (see Section 11.1) on the `ImageHandle` of the UEFI driver. If this process is completed, then `EFI_SUCCESS` is returned. If the resources are not available to complete the UEFI driver initialization, then `EFI_OUT_OF_RESOURCES` is returned.

**Status Codes Returned**

<table>
<thead>
<tr>
<th>Code</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>EFI_SUCCESS</code></td>
<td>The driver was initialized.</td>
</tr>
<tr>
<td><code>EFI_OUT_OF_RESOURCES</code></td>
<td>The request could not be completed due to a lack of resources.</td>
</tr>
</tbody>
</table>

### 4.2 EFI Table Header

The data type **EFI_TABLE_HEADER** is the data structure that precedes all of the standard EFI table types. It includes a signature that is unique for each table type, a revision of the table that may be updated as extensions are added to the EFI table types, and a 32-bit CRC so a consumer of an EFI table type can validate the contents of the EFI table.

**EFI_TABLE_HEADER**

**Summary**

Data structure that precedes all of the standard EFI table types.

**Related Definitions**

typedef struct {

}
```c
UINT64  Signature;
UINT32  Revision;
UINT32  HeaderSize;
UINT32  CRC32;
UINT32  Reserved;
} EFI_TABLE_HEADER;
```

**Parameters**

- **Signature**
  A 64-bit signature that identifies the type of table that follows. Unique signatures have been generated for the EFI System Table, the EFI Boot Services Table, and the EFI Runtime Services Table.

- **Revision**
  The revision of the EFI Specification to which this table conforms. The upper 16 bits of this field contain the major revision value, and the lower 16 bits contain the minor revision value. The minor revision values are binary coded decimals and are limited to the range of 00..99.

  When printed or displayed UEFI spec revision is referred as \((\text{Major revision}.)(\text{Minor revision upper decimal}).(\text{Minor revision lower decimal})\) or \((\text{Major revision}.)(\text{Minor revision upper decimal})\) in case Minor revision lower decimal is set to 0. For example:

  - A specification with the revision value \((2<<16) | (30)\) would be referred as 2.3;
  - A specification with the revision value \((2<<16) | (31)\) would be referred as 2.3.1

- **HeaderSize**
  The size, in bytes, of the entire table including the `EFI_TABLE_HEADER`.

- **CRC32**
  The 32-bit CRC for the entire table. This value is computed by setting this field to 0, and computing the 32-bit CRC for `HeaderSize` bytes.

- **Reserved**
  Reserved field that must be set to 0.

**Note:** The capabilities found in the EFI system table, runtime table and boot services table may change over time. The first field in each of these tables is an `EFI_TABLE_HEADER`. This header’s Revision field is incremented when new capabilities and functions are added to the functions in the table. When checking for capabilities, code should verify that Revision is greater than or equal to the revision level of the table at the point when the capabilities were added to the UEFI specification.

**Note:** Unless otherwise specified, UEFI uses a standard CCITT32 CRC algorithm with a seed polynomial value of 0x04c11db7 for its CRC calculations.

**Note:** The size of the system table, runtime services table, and boot services table may increase over time. It is very important to always use the `HeaderSize` field of the `EFI_TABLE_HEADER` to determine the size of these tables.

### 4.3 EFI System Table

UEFI uses the EFI System Table, which contains pointers to the runtime and boot services tables. The definition for this table is shown in the following code fragments. Except for the table header, all elements in the service tables are pointers to functions as defined in Section 7 and Section 8. Prior to a
call to EFI_BOOT_SERVICES.ExitBootServices(), all of the fields of the EFI System Table are valid. After an operating system has taken control of the platform with a call to ExitBootServices(), only theHdr, FirmwareVendor, FirmwareRevision, RuntimeServices, NumberOfTableEntries, and ConfigurationTable fields are valid.

EFI_SYSTEM_TABLE

Summary
Contains pointers to the runtime and boot services tables.

Related Definitions

```c
#define EFI_SYSTEM_TABLE_SIGNATURE   0x5453595320494249
#define EFI_2_90_SYSTEM_TABLE_REVISION ((2<<16) | (90))
#define EFI_2_80_SYSTEM_TABLE_REVISION ((2<<16) | (80))
#define EFI_2_70_SYSTEM_TABLE_REVISION ((2<<16) | (70))
#define EFI_2_60_SYSTEM_TABLE_REVISION ((2<<16) | (60))
#define EFI_2_50_SYSTEM_TABLE_REVISION ((2<<16) | (50))
#define EFI_2_40_SYSTEM_TABLE_REVISION ((2<<16) | (40))
#define EFI_2_31_SYSTEM_TABLE_REVISION ((2<<16) | (31))
#define EFI_2_30_SYSTEM_TABLE_REVISION ((2<<16) | (30))
#define EFI_2_20_SYSTEM_TABLE_REVISION ((2<<16) | (20))
#define EFI_2_10_SYSTEM_TABLE_REVISION ((2<<16) | (10))
#define EFI_2_00_SYSTEM_TABLE_REVISION ((2<<16) | (00))
#define EFI_1_10_SYSTEM_TABLE_REVISION ((1<<16) | (10))
#define EFI_1_02_SYSTEM_TABLE_REVISION ((1<<16) | (02))
#define EFI_SPECIFICATION_VERSION   EFI_SYSTEM_TABLE_REVISION
#define EFI_SYSTEM_TABLE_REVISION   EFI_2_90_SYSTEM_TABLE_REVISION

typedef struct {
    EFI_TABLE_HEADER            Hdr;
    CHAR16                      *FirmwareVendor;
    UINT32                      FirmwareRevision;
    EFI_HANDLE                  ConsoleInHandle;
    EFI_SIMPLE_TEXT_INPUT_PROTOCOL  *ConIn;
    EFI_HANDLE                  ConsoleOutHandle;
    EFI_SIMPLE_TEXT_OUTPUT_PROTOCOL *ConOut;
    EFI_HANDLE                  StandardErrorHandle;
    EFI_SIMPLE_TEXT_OUTPUT_PROTOCOL *StdErr;
    EFI_RUNTIME_SERVICES        *RuntimeServices;
    EFI_BOOT_SERVICES           *BootServices;
    UINTN                       NumberOfTableEntries;
    EFI_CONFIGURATION_TABLE    *ConfigurationTable;
} EFI_SYSTEM_TABLE;
```
Parameters

**Hdr**
The table header for the EFI System Table. This header contains the `EFI_SYSTEM_TABLE_SIGNATURE` and `EFI_SYSTEM_TABLE_REVISION` values along with the size of the `EFI_SYSTEM_TABLE` structure and a 32-bit CRC to verify that the contents of the EFI System Table are valid.

**FirmwareVendor**
A pointer to a null terminated string that identifies the vendor that produces the system firmware for the platform.

**FirmwareRevision**
A firmware vendor specific value that identifies the revision of the system firmware for the platform.

**ConsoleInHandle**
The handle for the active console input device. This handle must support `EFI_SIMPLE_TEXT_INPUT_PROTOCOL` and `EFI_SIMPLE_TEXT_INPUT_EX_PROTOCOL`. If there is no active console, these protocols must still be present.

**ConIn**
A pointer to the `EFI_SIMPLE_TEXT_INPUT_PROTOCOL` interface that is associated with `ConsoleInHandle`.

**ConsoleOutHandle**
The handle for the active console output device. This handle must support the `EFI_SIMPLE_TEXT_OUTPUT_PROTOCOL`. If there is no active console, this protocol must still be present.

**ConOut**
A pointer to the `EFI_SIMPLE_TEXT_OUTPUT_PROTOCOL` interface that is associated with `ConsoleOutHandle`.

**StandardErrorHandle**
The handle for the active standard error console device. This handle must support the `EFI_SIMPLE_TEXT_OUTPUT_PROTOCOL`. If there is no active console, this protocol must still be present.

**StdErr**
A pointer to the `EFI_SIMPLE_TEXT_OUTPUT_PROTOCOL` interface that is associated with `StandardErrorHandle`.

**RuntimeServices**
A pointer to the EFI Runtime Services Table. See Section 4.5.

**BootServices**
A pointer to the EFI Boot Services Table. See Section 4.4.

**NumberOfTableEntries**
The number of system configuration tables in the buffer `ConfigurationTable`.

**ConfigurationTable**
A pointer to the system configuration tables. The number of entries in the table is `NumberOfTableEntries`.

4.4 EFI Boot Services Table

UEFI uses the EFI Boot Services Table, which contains a table header and pointers to all of the boot services. The definition for this table is shown in the following code fragments. Except for the table header, all elements in the EFI Boot Services Tables are prototypes of function pointers to functions as defined in Section 7. The function pointers in this table are not valid after the operating system has taken control of the platform with a call to `EFI_BOOT_SERVICES.ExitBootServices()`.

**EFI_BOOT_SERVICES**

**Summary**
Contains a table header and pointers to all of the boot services.
Related Definitions

```c
#define EFI_BOOT_SERVICES_SIGNATURE   0x56524553544f4f42
#define EFI_BOOT_SERVICES_REVISION   EFI_SPECIFICATION_VERSION

typedef struct {
    EFI_TABLE_HEADER Hdr;

    // Task Priority Services
    EFI_RAISE_TPL    RaiseTPL;      // EFI 1.0+
    EFI_RESTORE_TPL  RestoreTPL;    // EFI 1.0+

    // Memory Services
    EFI_ALLOCATE_PAGES AllocatePages; // EFI 1.0+
    EFI_FREE_PAGES   FreePages;      // EFI 1.0+
    EFI_GET_MEMORY_MAP GetMemoryMap; // EFI 1.0+
    EFI_ALLOCATE_POOL AllocatePool; // EFI 1.0+
    EFI_FREE_POOL    FreePool;       // EFI 1.0+

    // Event & Timer Services
    EFI_CREATE_EVENT CreateEvent;   // EFI 1.0+
    EFI_SET_TIMER    SetTimer;       // EFI 1.0+
    EFI_WAIT_FOR_EVENT WaitForEvent; // EFI 1.0+
    EFI_SIGNAL_EVENT SignalEvent;   // EFI 1.0+
    EFI_CLOSE_EVENT  CloseEvent;    // EFI 1.0+
    EFI_CHECK_EVENT  CheckEvent;    // EFI 1.0+

    // Protocol Handler Services
    EFI_INSTALL_PROTOCOL_INTERFACE InstallProtocolInterface; // EFI 1.0+
    EFI_REINSTALL_PROTOCOL_INTERFACE ReinstallProtocolInterface; // EFI 1.0+
    EFI_UNINSTALL_PROTOCOL_INTERFACE UninstallProtocolInterface; // EFI 1.0+
    EFI_HANDLE_PROTOCOL  HandleProtocol;   // EFI 1.0+
    VOID* Reserved;       // EFI 1.0+
    EFI_REGISTER_PROTOCOL_NOTIFY RegisterProtocolNotify; // EFI 1.0+
    EFI_LOCATE_HANDLE    LocateHandle;    // EFI 1.0+
    EFI_LOCATE_DEVICE_PATH LocateDevicePath; // EFI 1.0+
    EFI_INSTALL_CONFIGURATION_TABLE InstallConfigurationTable; // EFI 1.0+
} EFI_BOOT_SERVICES;
```
// Image Services
EFI_IMAGE_LOAD          LoadImage;    // EFI 1.0+
EFI_IMAGE_START         StartImage;  // EFI 1.0+
EFI_EXIT                Exit;        // EFI 1.0+
EFI_IMAGE_UNLOAD        UnloadImage; // EFI 1.0+
EFI_EXIT_BOOT_SERVICES  ExitBootServices; // EFI 1.0+

// Miscellaneous Services
EFI_GET_NEXT_MONOTONIC_COUNT GetNextMonotonicCount; // EFI 1.0+
EFISTALL               Stall;        // EFI 1.0+
EFI_SET_WATCHDOG_TIMER  SetWatchdogTimer; // EFI 1.0+

// DriverSupport Services
EFI_CONNECT_CONTROLLER   ConnectController; // EFI 1.1
EFI_DISCONNECT_CONTROLLER DisconnectController; // EFI 1.1+

// Open and Close Protocol Services
EFI_OPEN_PROTOCOL        OpenProtocol;  // EFI 1.1+
EFI_CLOSE_PROTOCOL       CloseProtocol; // EFI 1.1+
EFI_OPEN_PROTOCOL_INFORMATION OpenProtocolInformation; // EFI 1.1+

// Library Services
EFI_PROTOCOLS_PER_HANDLE ProtocolsPerHandle; // EFI 1.1+
EFI_LOCATE_HANDLE_BUFFER LocateHandleBuffer; // EFI 1.1+
EFI_LOCATE_PROTOCOL     LocateProtocol; // EFI 1.1+
EFI_INSTALL_MULTIPLE_PROTOCOL_INTERFACES InstallMultipleProtocolInterfaces; // EFI 1.1+
EFI_UNINSTALL_MULTIPLE_PROTOCOL_INTERFACES UninstallMultipleProtocolInterfaces; // EFI 1.1+

// 32-bit CRC Services
EFI_CALCULATE_CRC32     CalculateCrc32; // EFI 1.1+
Miscellaneous Services

EFI_COPY_MEM CopyMem; // EFI 1.1+
EFI_SET_MEM SetMem; // EFI 1.1+
EFI_CREATE_EVENT_EX CreateEventEx; // UEFI 2.0+

Parameters

Hdr
The table header for the EFI Boot Services Table. This header contains the EFI_BOOT_SERVICES_SIGNATURE and EFI_BOOT_SERVICES_REVISION values along with the size of the EFI_BOOT_SERVICES structure and a 32-bit CRC to verify that the contents of the EFI Boot Services Table are valid.

RaiseTPL
Raising the task priority level.

RestoreTPL
Restores/lowers the task priority level.

AllocatePages
Allocates pages of a particular type.

FreePages
Frees allocated pages.

GetMemoryMap
Returns the current boot services memory map and memory map key.

AllocatePool
Allocates a pool of a particular type.

FreePool
Frees allocated pool.

CreateEvent
Creates a general-purpose event structure.

SetTimer
Sets an event to be signaled at a particular time.

WaitForEvent
Stops execution until an event is signaled.

SignalEvent
Signals an event.

CloseEvent
Closes and frees an event structure.

CheckEvent
Checks whether an event is in the signaled state.

InstallProtocolInterface
Installs a protocol interface on a device handle.

ReinstallProtocolInterface
Reinstalls a protocol interface on a device handle.

UninstallProtocolInterface
Removes a protocol interface from a device handle.

HandleProtocol
Queries a handle to determine if it supports a specified protocol.

Reserved
Reserved. Must be NULL.

RegisterProtocolNotify
Registers an event that is to be signaled whenever an interface is installed for a specified protocol.

LocateHandle
Returns an array of handles that support a specified protocol.

LocateDevicePath
Locates all devices on a device path that support a specified protocol and returns the handle to the device that is closest to the path.
InstallConfigurationTable
Adds, updates, or removes a configuration table from the EFI System Table.

LoadImage
Loads an EFI image into memory.

StartImage
Transfers control to a loaded image’s entry point.

Exit
Exits the image’s entry point.

UnloadImage
Unloads an image.

ExitBootServices
Terminates boot services.

GetNextMonotonicCount
Returns a monotonically increasing count for the platform.

Stall
Stalls the processor.

SetWatchdogTimer
Resets and sets a watchdog timer used during boot services time.

ConnectController
Uses a set of precedence rules to find the best set of drivers to manage a controller.

DisconnectController
Informs a set of drivers to stop managing a controller.

OpenProtocol
Adds elements to the list of agents consuming a protocol interface.

CloseProtocol
Removes elements from the list of agents consuming a protocol interface.

OpenProtocolInformation
Retrieve the list of agents that are currently consuming a protocol interface.

ProtocolsPerHandle
Retrieves the list of protocols installed on a handle. The return buffer is automatically allocated.

LocateHandleBuffer
Retrieves the list of handles from the handle database that meet the search criteria. The return buffer is automatically allocated.

LocateProtocol
Finds the first handle in the handle database that supports the requested protocol.

InstallMultipleProtocolInterfaces
Installs one or more protocol interfaces onto a handle.

UninstallMultipleProtocolInterfaces
Uninstalls one or more protocol interfaces from a handle.

CalculateCrc32
Computes and returns a 32-bit CRC for a data buffer.

CopyMem
Copies the contents of one buffer to another buffer.

SetMem
Fills a buffer with a specified value.

CreateEventEx
Creates an event structure as part of an event group.

4.5 EFI Runtime Services Table
UEFI uses the EFI Runtime Services Table, which contains a table header and pointers to all of the runtime services. The definition for this table is shown in the following code fragments. Except for the table header, all elements in the EFI Runtime Services Tables are prototypes of function pointers to functions as defined in Section 8. Unlike the EFI Boot Services Table, this table, and the function pointers it contains
are valid after the UEFI OS loader and OS have taken control of the platform with a call to 
EFI_BOOT_SERVICES.ExitBootServices(). If a call to SetVirtualAddressMap() is made by the 
OS, then the function pointers in this table are fixed up to point to the new virtually mapped entry points.

**EFI_RUNTIME_SERVICES**

**Summary**

Contains a table header and pointers to all of the runtime services.

**Related Definitions**

```
#define EFI_RUNTIME_SERVICES_SIGNATURE 0x56524553544e5552
#define EFI_RUNTIME_SERVICES_REVISION EFI_SPECIFICATION_VERSION
typedef struct {
    EFI_TABLE_HEADER                  Hdr;

    //
    // Time Services
    //
    EFI_GET_TIME                        GetTime;
    EFI_SET_TIME                        SetTime;
    EFI_GET_WAKEUP_TIME                 GetWakeupTime;
    EFI_SET_WAKEUP_TIME                 SetWakeupTime;

    //
    // Virtual Memory Services
    //
    EFI_SET_VIRTUAL_ADDRESS_MAP         SetVirtualAddressMap;
    EFI_CONVERT_POINTER                ConvertPointer;

    //
    // Variable Services
    //
    EFI_GET_VARIABLE                    GetVariable;
    EFI_GET_NEXT_VARIABLE_NAME         GetNextVariableName;
    EFI_SET_VARIABLE                    SetVariable;

    //
    // Miscellaneous Services
    //
    EFI_GET_NEXT_HIGH_MONO_COUNT       GetNextHighMonotonicCount;
    EFI_RESET_SYSTEM                   ResetSystem;

    //
    // UEFI 2.0 Capsule Services
    //
```
```c
EFI_UPDATE_CAPSULE          UpdateCapsule;
EFI_QUERY_CAPSULE_CAPABILITIES QueryCapsuleCapabilities;

// Miscellaneous UEFI 2.0 Service
//
EFI_QUERY_VARIABLE_INFO  QueryVariableInfo;
} EFI_RUNTIME_SERVICES;
```

**Parameters**

- **Hdr**
  The table header for the EFI Runtime Services Table. This header contains the `EFI_RUNTIME_SERVICES_SIGNATURE` and `EFI_RUNTIME_SERVICES_REVISION` values along with the size of the `EFI_RUNTIME_SERVICES` structure and a 32-bit CRC to verify that the contents of the EFI Runtime Services Table are valid.

- **GetTime**
  Returns the current time and date, and the time-keeping capabilities of the platform.

- **SetTime**
  Sets the current local time and date information.

- **GetWakeupTime**
  Returns the current wakeup alarm clock setting.

- **SetWakeupTime**
  Sets the system wakeup alarm clock time.

- **SetVirtualAddressMap**
  Used by a UEFI OS loader to convert from physical addressing to virtual addressing.

- **ConvertPointer**
  Used by EFI components to convert internal pointers when switching to virtual addressing.

- **GetVariable**
  Returns the value of a variable.

- **GetNextVariableName**
  Enumerates the current variable names.

- **SetVariable**
  Sets the value of a variable.

- **GetNextHighMonotonicCount**
  Returns the next high 32 bits of the platform’s monotonic counter.

- **ResetSystem**
  Resets the entire platform.

- **UpdateCapsule**
  Passes capsules to the firmware with both virtual and physical mapping.

- **QueryCapsuleCapabilities**
  Returns if the capsule can be supported via `UpdateCapsule()`.

- **QueryVariableInfo**
  Returns information about the EFI variable store.

### 4.6 EFI Configuration Table & Properties Table

The EFI Configuration Table is the `ConfigurationTable` field in the EFI System Table. This table contains a set of GUID(pointer pairs. Each element of this table is described by the `EFI_CONFIGURATION_TABLE` structure below. The number of types of configuration tables is expected to grow over time. This is why a GUID is used to identify the configuration table type. The EFI Configuration Table may contain at most once instance of each table type.
EFI_CONFIGURATION_TABLE

Summary
Contains a set of GUID/pointer pairs comprised of the ConfigurationTable field in the EFI System Table.

Related Definitions
```c
typedef struct{
    EFI_GUID VendorGuid;
    VOID *VendorTable;
} EFI_CONFIGURATION_TABLE;
```

Parameters
- **VendorGuid**: The 128-bit GUID value that uniquely identifies the system configuration table.
- **VendorTable**: A pointer to the table associated with VendorGuid. Type of the memory that is used to store the table as well as whether this pointer is a physical address or a virtual address during runtime (whether or not a particular address reported in the table gets fixed up when a call to SetVirtualAddressMap() is made) is determined by the VendorGuid. Unless otherwise specified, memory type of the table buffer is defined by the guidelines set forth in the Calling Conventions section in Chapter 2. It is the responsibility of the specification defining the VendorTable to specify additional memory type requirements (if any) and whether to convert the addresses reported in the table. Any required address conversion is a responsibility of the driver that publishes corresponding configuration table.

Industry Standard Configuration Tables
The following list shows the GUIDs for tables defined in some of the industry standards. These industry standards define tables accessed as UEFI Configuration Tables on UEFI-based systems. All the addresses reported in these table entries will be referenced as physical and will not be fixed up when transition from preboot to runtime phase. This list is not exhaustive and does not show GUIDs for all possible UEFI Configuration tables.
#define EFI_ACPI_20_TABLE_GUID \   {0x8868e871,0xe4f1,0x11d3,\   {0xbc,0x22,0x00,0x80,0xc7,0x3c,0x88,0x81}}

#define ACPI_TABLE_GUID \   {0xeb9d2d30,0x2d88,0x11d3,\   {0x9a,0x16,0x00,0x90,0x27,0x3f,0xc1,0x4d}}

#define SAL_SYSTEM_TABLE_GUID \   {0xeb9d2d32,0x2d88,0x11d3,\   {0x9a,0x16,0x00,0x90,0x27,0x3f,0xc1,0x4d}}

#define SMBIOS_TABLE_GUID \   {0xeb9d2d31,0x2d88,0x11d3,\   {0x9a,0x16,0x00,0x90,0x27,0x3f,0xc1,0x4d}}

#define SMBIOS3_TABLE_GUID \   {0xf2fd1544, 0x9794, 0x4a2c,\   {0x99,0x2e,0xe5,0xbb,0xcf,0x20,0xe3,0x94})

#define MPS_TABLE_GUID \   {0xeb9d2d3f,0x2d88,0x11d3,\   {0x9a,0x16,0x00,0x90,0x27,0x3f,0xc1,0x4d}}

  // ACPI 2.0 or newer tables should use EFI_ACPI_TABLE_GUID
  //
  #define EFI_ACPI_TABLE_GUID \   {0x8868e871,0xe4f1,0x11d3,\   {0xbc,0x22,0x00,0x80,0xc7,0x3c,0x88,0x81}}

#define EFI_ACPI_20_TABLE_GUID EFI_ACPI_TABLE_GUID
#define ACPI_TABLE_GUID ACPI_TABLE_GUID

#define EFI_ACPI_10_TABLE_GUID ACPI_TABLE_GUID

JSON Configuration Tables

The following list shows the GUIDs for tables defined for reporting firmware configuration data to EFI Configuration Tables and also for processing JSON payload capsule as defined in Section 23.5. The address reported in the table entry identified by EFI_JSON_CAPSULE_DATA_TABLE_GUID will be referenced as physical and will not be fixed up when transition from preboot to runtime phase. The addresses reported in these table entries identified by EFI_JSON_CONFIG_DATA_TABLE_GUID and EFI_JSON_CAPSULE_RESULT_TABLE_GUID will be referenced as virtual and will be fixed up when transition from preboot to runtime phase.
#define EFI_JSON_CONFIG_DATA_TABLE_GUID \
{0x87367f87, 0x1119, 0x41ce, \ 
{0xaa, 0xec, 0xb8, 0xe0, 0x11, 0x1f, 0x55, 0x8a }}

#define EFI_JSON_CAPSULE_DATA_TABLE_GUID \
{0x35e7a725, 0x8dd2, 0x4cac, \ 
{ 0x80, 0x11, 0x33, 0xcd, 0xa8, 0x10, 0x90, 0x56 }}

#define EFI_JSON_CAPSULE_RESULT_TABLE_GUID \
{0xdbc461c3, 0xb3de, 0x422a,\ 
{0xb9, 0xb4, 0x98, 0x86, 0xfd, 0x49, 0xa1, 0xe5 }}

**Devicetree Tables**

The following list shows the GUIDs for the Devicetree table (DTB). For more information, see “Links to UEFI-Related Documents” ([http://uefi.org/uefi](http://uefi.org/uefi)) under the headings “Devicetree Specification”. The DTB must be contained in memory of type EfiACPIReclaimMemory. The address reported in this table entry will be referenced as physical and will not be fixed up when transition from preboot to runtime phase.

Firmware must have the DTB resident in memory and installed in the EFI system table before executing any UEFI applications or drivers that are not part of the system firmware image. Once the DTB is installed as a configuration table, the system firmware must not make any modification to it or reference any data contained within the DTB.

UEFI applications are permitted to modify or replace the loaded DTB. System firmware must not depend on any data contained within the DTB. If system firmware makes use of a DTB for its own configuration, it should use a separate private copy that is not installed in the EFI System Table or otherwise be exposed to EFI applications.

```c
// Devicetree table, in Flattened Devicetree Blob (DTB) format
//
#define EFI_DTB_TABLE_GUID \
{0xb1b621d5, 0xf19c, 0x41a5, \ 
{0x83, 0x0b, 0xd9, 0x15, 0x2c, 0x69, 0xaa, 0xe0}}
```

**EFI_RT_PROPERTIES_TABLE**

This table should be published by a platform if it no longer supports all EFI runtime services once ExitBootServices() has been called by the OS. Note that this is merely a hint to the OS, which it is free to ignore, and so the platform is still required to provide callable implementations of unsupported runtime services that simply return EFI_UNSUPPORTED.
#define EFI_RT_PROPERTIES_TABLE_GUID\
{ 0xeb66918a, 0x7eef, 0x402a, \\{ 0x84, 0x2e, 0x93, 0x1d, 0x21, 0xc3, 0x8a, 0xe9 }}

typedef struct {
    UINT16 Version;
    UINT16 Length;
    UINT32 RuntimeServicesSupported;
} EFI_RT_PROPERTIES_TABLE;

**Version**
Version of the table, must be 0x1

#define EFI_RT_PROPERTIES_TABLE_VERSION 0x1

**Length**
Size in bytes of the entire EFI_RT_PROPERTIES_TABLE, must be 8.

**RuntimeServicesSupported**
Bitmask of which calls are or are not supported, where a bit set to 1 indicates that the call is supported, and 0 indicates that it is not.

#define EFI_RT_SUPPORTED_GET_TIME 0x0001
#define EFI_RT_SUPPORTED_SET_TIME 0x0002
#define EFI_RT_SUPPORTED_GET_WAKEUP_TIME 0x0004
#define EFI_RT_SUPPORTED_SET_WAKEUP_TIME 0x0008
#define EFI_RT_SUPPORTED_GET_VARIABLE 0x0010
#define EFI_RT_SUPPORTED_GET_NEXT_VARIABLE_NAME 0x0020
#define EFI_RT_SUPPORTED_SET_VARIABLE 0x0040
#define EFI_RT_SUPPORTED_SET_VIRTUAL_ADDRESS_MAP 0x0080
#define EFI_RT_SUPPORTED_CONVERT_POINTER 0x0100
#define EFI_RT_SUPPORTED_GET_NEXT_HIGH_MONOTONIC_COUNT 0x0200
#define EFI_RT_SUPPORTED_RESET_SYSTEM 0x0400
#define EFI_RT_SUPPORTED_UPDATE_CAPSULE 0x0800
#define EFI_RT_SUPPORTED_QUERY_CAPSULE_CAPABILITIES 0x1000
#define EFI_RT_SUPPORTED_QUERY_VARIABLE_INFO 0x2000

The address reported in the EFI configuration table entry of this type will be referenced as physical and will not be fixed up when transitioning from preboot to runtime phase.

**EFI_PROPERTIES_TABLE (deprecated)**

Note: This table is deprecated and should no longer be used! It will be removed from future versions of the specification. EFI_MEMORY_ATTRIBUTES_TABLE described below provides alternative mechanism to implement runtime memory protection.

This table is published if the platform meets some of the construction requirements listed in the MemoryProtectionAttributes.
typedef struct {
    UINT32  Version;
    UINT32  Length;
    UINT64  MemoryProtectionAttribute;
} EFI_PROPERTIES_TABLE;

Version
This is revision of the table. Successive version may populate additional bits and growth the table length. In the case of the latter, the Length field will be adjusted appropriately.

#define EFI_PROPERTIES_TABLE_VERSION 0x00010000

Length
This is the size of the entire EFI_PROPERTIES_TABLE structure, including the version. The initial version will be of length 16.

MemoryProtectionAttribute
This field is a bit mask. Any bits not defined shall be considered reserved. A set bit means that the underlying firmware has been constructed responsive to the given property.

    //
    // Memory attribute (Not defined bits are reserved)
    //
    #define EFI_PROPERTIES_RUNTIME_MEMORY_PROTECTION_NON_EXECUTABLE_PE_DATA 0x1
    
    // BIT 0 – description – implies the runtime data is separated from the code

This bit implies that the UEFI runtime code and data sections of the executable image are separate and must be aligned as specified in Section 2.3. This bit also implies that the data pages do not have any executable code.

It is recommended not to use this attribute, especially for implementations that broke the runtime code memory map descriptors into the underlying code and data sections within UEFI modules. This splitting causes interoperability issues with operating systems that invoke SetVirtualAddress() without realizing that there is a relationship between these runtime descriptors.

EFI_MEMORY_ATTRIBUTES_TABLE

Summary
When published by the system firmware, the EFI_MEMORY_ATTRIBUTES_TABLE provides additional information about regions within the run-time memory blocks defined in the EFI_MEMORY_DESCRIPTOR entries returned from EFI_BOOT_SERVICES.GetMemoryMap() function. The Memory Attributes Table is currently used to describe memory protections that may be applied to the EFI Runtime code and data by an operating system or hypervisor. Consumers of this table must currently ignore entries containing any values for Type except for EfiRuntimeServicesData and EfiRuntimeServicesCode to ensure compatibility with future uses of this table. The Memory Attributes Table may define multiple entries to describe sub-regions that comprise a single entry returned by GetMemoryMap() however the sub-regions must total to completely describe the larger
region and may not cross boundaries between entries reported by \texttt{GetMemoryMap()}. If a run-time region returned in GetMemoryMap() entry is not described within the Memory Attributes Table, this region is assumed to not be compatible with any memory protections.

Only entire \texttt{EFI_MEMORY_DESCRIPTOR} entries as returned by \texttt{GetMemoryMap()} may be passed to \texttt{SetVirtualAddressMap()}. The address reported in the EFI configuration table entry of this type will be referenced as physical and will not be fixed up when transition from preboot to runtime phase.

Prototype

\begin{verbatim}
#define EFI_MEMORY_ATTRIBUTES_TABLE_GUID \\
{ 0xdcfa911d, 0x26eb, 0x469f, \\
 {0xa2, 0x20, 0x38, 0xb7, 0xdc, 0x46, 0x12, 0x20}}
\end{verbatim}

With the following data structure

\begin{verbatim}
/***********************************************/
/* EFI_MEMORY_ATTRIBUTES_TABLE */
/***********************************************/
typedef struct {
    UINT32 Version ;
    UINT32 NumberOfEntries ;
    UINT32 DescriptorSize ;
    UINT32 Reserved ;
    // EFI_MEMORY_DESCRIPTOR Entry [1];
} EFI_MEMORY_ATTRIBUTES_TABLE;
\end{verbatim}

\begin{itemize}
 \item **Version** The version of this table. Present version is 0x00000001
 \item **NumberOfEntries** Count of \texttt{EFI_MEMORY_DESCRIPTOR} entries provided. This is typically the total number of PE/COFF sections within all UEFI modules that comprise the UEFI Runtime and all UEFI Runtime Data regions (e.g. runtime heap).
 \item **Entry** Array of Entries of type \texttt{EFI_MEMORY_DESCRIPTOR}.
 \item **DescriptorSize** Size of the memory descriptor.
 \item **Reserved** Reserved bytes.
\end{itemize}

Description

For each array entry, the \texttt{EFI_MEMORY_DESCRIPTOR. Attribute} field can inform a runtime agency, such as operating system or hypervisor, as to what class of protection settings can be made in the memory management unit for the memory defined by this entry. The only valid bits for \texttt{Attribute} field currently are \texttt{EFI_MEMORY_RO, EFI_MEMORY_XP}, plus \texttt{EFI_MEMORY_RUNTIME}. Irrespective of the memory protections implied by \texttt{Attribute}, the
**EFI_MEMORY_DESCRIPTOR**. Type field should match the type of the memory in enclosing `SetMemoryMap()` entry. PhysicalStart must be aligned as specified in Section 2.3. The list must be sorted by physical start address in ascending order. VirtualStart field must be zero and ignored by the OS since it has no purpose for this table. NumPages must cover the entire memory region for the protection mapping. Each Descriptor in the **EFI_MEMORY_ATTRIBUTES_TABLE** with attribute **EFI_MEMORY_RUNTIME** must not overlap any other Descriptor in the **EFI_MEMORY_ATTRIBUTES_TABLE** with attribute **EFI_MEMORY_RUNTIME**. Additionally, every memory region described by a Descriptor in **EFI_MEMORY_ATTRIBUTES_TABLE** must be a sub-region of, or equal to, a descriptor in the table produced by `GetMemoryMap()`.

Table 4-1 Usage of Memory Attribute Definitions

<table>
<thead>
<tr>
<th></th>
<th>EFI_MEMORY_RO</th>
<th>EFI_MEMORY_XP</th>
<th>EFI_MEMORY_RUNTIME</th>
</tr>
</thead>
<tbody>
<tr>
<td>No memory access</td>
<td>0</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>protection is possible for Entry</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Write-protected Code</td>
<td>1</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>Read/Write Data</td>
<td>0</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>Read-only Data</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
</tbody>
</table>

Other Configuration Tables

The following list shows additional configuration tables defined in this specification:

- **EFI_MEMORY_RANGE_CAPSULE_GUID**
- **EFI_DEBUG_IMAGE_INFO_TABLE** (Section 18.4.3)
- **EFI_SYSTEM_RESOURCE_TABLE** (Section 23.4)
- **EFI_IMAGE_EXECUTION_INFO_TABLE** (Section 32.5.3.1)
- User Information Table (Section 36.5)
- HII Database export buffer (Section 33.2.11.1, OS Runtime Utilization)

4.7 Image Entry Point Examples

The examples in the following sections show how the various table examples are presented in the UEFI environment.

4.7.1 Image Entry Point Examples

The following example shows the image entry point for a UEFI Application. This application makes use of the EFI System Table, the EFI Boot Services Table, and the EFI Runtime Services Table.

```c
EFI_SYSTEM_TABLE *gST;
```
EfiApplicationEntryPoint(
    IN EFI_HANDLE ImageHandle,
    IN EFI_SYSTEM_TABLE *SystemTable
) {
    EFI_STATUS Status;
    EFI_TIME *Time;

    gST = SystemTable;
    gBS = gST->BootServices;
    gRT = gST->RuntimeServices;

    // Use EFI System Table to print “Hello World” to the active console output
    // device.
    Status = gST->ConOut->OutputString (gST->ConOut, L"Hello World\n\r");
    if (EFI_ERROR (Status)) {
        return Status;
    }

    // Use EFI Boot Services Table to allocate a buffer to store the current time
    // and date.
    Status = gBS->AllocatePool (EfiBootServicesData,
                          sizeof (EFI_TIME),
                          (VOID **)&Time);
    if (EFI_ERROR (Status)) {
        return Status;
    }

    // Use the EFI Runtime Services Table to get the current time and date.
    Status = gRT->GetTime (Time, NULL)
    if (EFI_ERROR (Status)) {
        return Status;
    }

    return Status;
}

The following example shows the UEFI image entry point for a driver that does not follow the UEFI Driver Model. Since this driver returns EFI_SUCCESS, it will stay resident in memory after it exits.
```
EFI_RUNTIME_SERVICES    *gRT;

EfiDriverEntryPoint(
    IN EFI_HANDLE   ImageHandle,
    IN EFI_SYSTEM_TABLE  *SystemTable
)
{
    gST = SystemTable;
    gBS = gST->BootServices;
    gRT = gST->RuntimeServices;

    //
    // Implement driver initialization here.
    //
    return EFI_SUCCESS;
}
```

The following example shows the UEFI image entry point for a driver that also does not follow the UEFI Driver Model. Since this driver returns EFI_DEVICE_ERROR, it will not stay resident in memory after it exits.

```
EFI_SYSTEM_TABLE        *gST;
EFI_BOOT_SERVICES       *gBS;
EFI_RUNTIME_SERVICES    *gRT;

EfiDriverEntryPoint(
    IN EFI_HANDLE   ImageHandle,
    IN EFI_SYSTEM_TABLE  *SystemTable
)
{
    gST = SystemTable;
    gBS = gST->BootServices;
    gRT = gST->RuntimeServices;

    //
    // Implement driver initialization here.
    //
    return EFI_DEVICE_ERROR;
}
```

### 4.7.2 UEFI Driver Model Example

The following is an UEFI Driver Model example that shows the driver initialization routine for the ABC device controller that is on the XYZ bus. The EFI DRIVER_BINDING_PROTOCOL and the function prototypes for AbcSupported(), AbcStart(), and AbcStop() are defined in Section 11.1. This function saves the driver's image handle and a pointer to the EFI boot services table in global variables, so the other functions in the same driver can have access to these values. It then creates an instance of the EFI DRIVER_BINDING_PROTOCOL and installs it onto the driver's image handle.
extern EFI_GUID gEfiDriverBindingProtocolGuid;
EFI_BOOT_SERVICES *gBS;
static EFI_DRIVER_BINDING_PROTOCOL mAbcDriverBinding = {
  AbcSupported,
  AbcStart,
  AbcStop,
  1,
  NULL,
  NULL
};

AbcEntryPoint(
  IN EFI_HANDLE ImageHandle,
  IN EFI_SYSTEM_TABLE *SystemTable
)
{
  EFI_STATUS Status;
  gBS = SystemTable->BootServices;
  mAbcDriverBinding->ImageHandle = ImageHandle;
  mAbcDriverBinding->DriverBindingHandle = ImageHandle;
  Status = gBS->InstallMultipleProtocolInterfaces(
    &mAbcDriverBinding->DriverBindingHandle,
    &gEfiDriverBindingProtocolGuid, &mAbcDriverBinding,
    NULL
  );
  return Status;
}

4.7.3 UEFI Driver Model Example (Unloadable)

The following is the same UEFI Driver Model example as above, except it also includes the code required to allow the driver to be unloaded through the boot service Unload(). Any protocols installed or memory allocated in AbcEntryPoint() must be uninstalled or freed in the AbcUnload().

extern EFI_GUID gEfiLoadedImageProtocolGuid;
extern EFI_GUID gEfiDriverBindingProtocolGuid;
EFI_BOOT_SERVICES *gBS;
static EFI_DRIVER_BINDING_PROTOCOL mAbcDriverBinding = {
  AbcSupported,
  AbcStart,
  AbcStop,
  1,
  NULL,
  NULL
};

EFI_STATUS AbcUnload(
  IN EFI_HANDLE ImageHandle
);
AbcEntryPoint(
    IN EFI_HANDLE ImageHandle,
    IN EFI_SYSTEM_TABLE *SystemTable
)
{
    EFI_STATUS Status;
    EFI_LOADED_IMAGE_PROTOCOL *LoadedImage;

    gBS = SystemTable->BootServices;

    Status = gBS->OpenProtocol(
        ImageHandle,
        &gEfiLoadedImageProtocolGuid,
        &LoadedImage,
        ImageHandle,
        NULL,
        EFI_OPEN_PROTOCOL_GET_PROTOCOL);
    if (EFI_ERROR (Status)) {
        return Status;
    }
    LoadedImage->Unload = AbcUnload;

    mAbcDriverBinding->ImageHandle = ImageHandle;
    mAbcDriverBinding->DriverBindingHandle = ImageHandle;

    Status = gBS->InstallMultipleProtocolInterfaces(
        &mAbcDriverBinding->DriverBindingHandle,
        &gEfiDriverBindingProtocolGuid, &mAbcDriverBinding,
        NULL);
    return Status;
}

EFI_STATUS AbcUnload(
    IN EFI_HANDLE ImageHandle
)
{
    EFI_STATUS Status;

    Status = gBS->UninstallMultipleProtocolInterfaces(
        ImageHandle,
        &gEfiDriverBindingProtocolGuid, &mAbcDriverBinding,
        NULL);
    return Status;
}
4.7.4 EFI Driver Model Example (Multiple Instances)

The following is the same as the first UEFI Driver Model example, except it produces three EFI DRIVER_BINDING_PROTOCOL instances. The first one is installed onto the driver’s image handle. The other two are installed onto newly created handles.

```c
extern EFI_GUID                    gEfiDriverBindingProtocolGuid;
EFI_BOOT_SERVICES                  *gBS;

static EFI_DRIVER_BINDING_PROTOCOL mAbcDriverBindingA = {
        AbcSupportedA,
        AbcStartA,
        AbcStopA,
        1,
        NULL,
        NULL
    };

static EFI_DRIVER_BINDING_PROTOCOL mAbcDriverBindingB = {
        AbcSupportedB,
        AbcStartB,
        AbcStopB,
        1,
        NULL,
        NULL
    };

static EFI_DRIVER_BINDING_PROTOCOL mAbcDriverBindingC = {
        AbcSupportedC,
        AbcStartC,
        AbcStopC,
        1,
        NULL,
        NULL
    };

AbcEntryPoint(
        IN EFI_HANDLE    ImageHandle,
        IN EFI_SYSTEM_TABLE *SystemTable
    )
{
    EFI_STATUS Status;

    gBS = SystemTable->BootServices;

    // // Install mAbcDriverBindingA onto ImageHandle
    // mAbcDriverBindingA->ImageHandle = ImageHandle;
    // mAbcDriverBindingA->DriverBindingHandle = ImageHandle;

    Status = gBS->InstallMultipleProtocolInterfaces(
        &mAbcDriverBindingA->DriverBindingHandle,
```
&gEfiDriverBindingProtocolGuid, &mAbcDriverBindingA,
    NULL
);  
if (EFI_ERROR (Status)) {
    return Status;
}

// Install mAbcDriverBindingB onto a newly created handle
//
mAbcDriverBindingB->ImageHandle     = ImageHandle;
mAbcDriverBindingB->DriverBindingHandle = NULL;

Status = gBS->InstallMultipleProtocolInterfaces(
    &mAbcDriverBindingB->DriverBindingHandle,
    &gEfiDriverBindingProtocolGuid, &mAbcDriverBindingB,
    NULL
);
if (EFI_ERROR (Status)) {
    return Status;
}

// Install mAbcDriverBindingC onto a newly created handle
//
mAbcDriverBindingC->ImageHandle     = ImageHandle;
mAbcDriverBindingC->DriverBindingHandle = NULL;

Status = gBS->InstallMultipleProtocolInterfaces(
    &mAbcDriverBindingC->DriverBindingHandle,
    &gEfiDriverBindingProtocolGuid, &mAbcDriverBindingC,
    NULL
);

return Status;
5 - GUID Partition Table (GPT) Disk Layout

5.1 GPT and MBR disk layout comparison

This specification defines the GUID Partition table (GPT) disk layout (i.e., partitioning scheme). The following list outlines the advantages of using the GPT disk layout over the legacy Master Boot Record (MBR) disk layout:

- Logical Block Addresses (LBAs) are 64 bits (rather than 32 bits).
- Supports many partitions (rather than just four primary partitions).
- Provides both a primary and backup partition table for redundancy.
- Uses version number and size fields for future expansion.
- Uses CRC32 fields for improved data integrity.
- Defines a GUID for uniquely identifying each partition.
- Uses a GUID and attributes to define partition content type.
- Each partition contains a 36 character human readable name.

5.2 LBA 0 Format

LBA 0 (i.e., the first logical block) of the hard disk contains either

- a legacy Master Boot Record (MBR) (see Section 5.2.1)
- or a protective MBR (see Section 5.2.3).

5.2.1 Legacy Master Boot Record (MBR)

A legacy MBR may be located at LBA 0 (i.e., the first logical block) of the disk if it is not using the GPT disk layout (i.e., if it is using the MBR disk layout). The boot code on the MBR is not executed by UEFI firmware.

Table 5-1 Legacy MBR

<table>
<thead>
<tr>
<th>Mnemonic</th>
<th>Byte Offset</th>
<th>Byte Length</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>BootCode</td>
<td>0</td>
<td>424</td>
<td>x86 code used on a non-UEFI system to select an MBR partition record and load the first logical block of that partition. This code shall not be executed on UEFI systems.</td>
</tr>
<tr>
<td>UniqueMBRdiskSignature</td>
<td>440</td>
<td>4</td>
<td>Unique Disk Signature This may be used by the OS to identify the disk from other disks in the system. This value is always written by the OS and is never written by EFI firmware.</td>
</tr>
<tr>
<td>Unknown</td>
<td>444</td>
<td>2</td>
<td>Unknown. This field shall not be used by UEFI firmware.</td>
</tr>
<tr>
<td>PartitionRecord</td>
<td>446</td>
<td>16*4</td>
<td>Array of four legacy MBR partition records (see Table 5-2).</td>
</tr>
</tbody>
</table>
The MBR contains four partition records (see Table 11) that each define the beginning and ending LBAs that a partition consumes on a disk.

**Table 5-2 Legacy MBR Partition Record**

<table>
<thead>
<tr>
<th>Mnemonic</th>
<th>Byte Offset</th>
<th>Byte Length</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>BootIndicator</td>
<td>0</td>
<td>1</td>
<td>0x80 indicates that this is the bootable legacy partition. Other values indicate that this is not a bootable legacy partition. This field shall not be used by UEFI firmware.</td>
</tr>
<tr>
<td>StartingCHS</td>
<td>1</td>
<td>3</td>
<td>Start of partition in CHS address format. This field shall not be used by UEFI firmware.</td>
</tr>
<tr>
<td>OSType</td>
<td>4</td>
<td>1</td>
<td>Type of partition. See Section 5.2.2.</td>
</tr>
<tr>
<td>EndingCHS</td>
<td>5</td>
<td>3</td>
<td>End of partition in CHS address format. This field shall not be used by UEFI firmware.</td>
</tr>
<tr>
<td>StartingLBA</td>
<td>8</td>
<td>4</td>
<td>Starting LBA of the partition on the disk. This field is used by UEFI firmware to determine the start of the partition.</td>
</tr>
<tr>
<td>SizeInLBA</td>
<td>12</td>
<td>4</td>
<td>Size of the partition in LBA units of logical blocks. This field is used by UEFI firmware to determine the size of the partition.</td>
</tr>
</tbody>
</table>

If an MBR partition has an **OSType** field of 0xEF (i.e., UEFI System Partition), then the firmware must add the UEFI System Partition GUID to the handle for the MBR partition using **InstallProtocolInterface()**. This allows drivers and applications, including OS loaders, to easily search for handles that represent UEFI System Partitions.

The following test must be performed to determine if a legacy MBR is valid:

- The Signature must be 0xaa55.
- A Partition Record that contains an **OSType** value of zero or a **SizeInLBA** value of zero may be ignored.

Otherwise:
- The partition defined by each MBR Partition Record must physically reside on the disk (i.e., not exceed the capacity of the disk).
- Each partition must not overlap with other partitions.

**Figure 5-1** shows an example of an MBR disk layout with four partitions.
Figure 5-1 MBRDisk Layout with legacy MBR example

Related Definitions:

```c
#pragma pack(1)

///
/// MBR Partition Entry
///
typedef struct {
    UINT8 BootIndicator;
    UINT8 StartHead;
    UINT8 StartSector;
    UINT8 StartTrack;
    UINT8 OSIndicator;
    UINT8 EndHead;
    UINT8 EndSector;
    UINT8 EndTrack;
    UINT8 StartingLBA[4];
    UINT8 SizeInLBA[4];
} MBR_PARTITION_RECORD;

///
/// MBR Partition Table
///
typedef struct {
    UINT8 BootStrapCode[440];
    UINT8 UniqueMbrSignature[4];
    UINT8 Unknown[2];
    MBR_PARTITION_RECORD Partition[4];
    UINT16 Signature;
} MASTER_BOOT_RECORD;

#pragma pack()
```
5.2.2 OS Types

Unique types defined by this specification (other values are not defined by this specification):

- 0xEF (i.e., UEFI System Partition) defines a UEFI system partition.
- 0xEE (i.e., GPT Protective) is used by a protective MBR (see 5.2.2) to define a fake partition covering the entire disk.

Other values are used by legacy operating systems, and are allocated independently of the UEFI specification.

*Note:* “Partition types” by Andries Brouwer: See “Links to UEFI-Related Documents” ([http://uefi.org/uefi](http://uefi.org/uefi)) under the heading “OS Type values used in the MBR disk layout”.

5.2.3 Protective MBR

For a bootable disk, a Protective MBR must be located at LBA 0 (i.e., the first logical block) of the disk if it is using the GPT disk layout. The Protective MBR precedes the GUID Partition Table Header to maintain compatibility with existing tools that do not understand GPT partition structures.

### Table 5-3 Protective MBR

<table>
<thead>
<tr>
<th>Mnemonic</th>
<th>Byte Offset</th>
<th>Byte Length</th>
<th>Contents</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Boot Code</strong></td>
<td>0</td>
<td>440</td>
<td>Unused by UEFI systems.</td>
</tr>
<tr>
<td><strong>Unique MBR Disk Signature</strong></td>
<td>440</td>
<td>4</td>
<td>Unused. Set to zero.</td>
</tr>
<tr>
<td><strong>Unknown</strong></td>
<td>444</td>
<td>2</td>
<td>Unused. Set to zero.</td>
</tr>
</tbody>
</table>
| **Partition Record**      | 446         | 16*4        | Array of four MBR partition records. Contains:
|                           |             |             |   • one partition record as defined Table 5-4; and                     |
|                           |             |             |   • three partition records each set to zero.                           |
| **Signature**             | 510         | 2           | Set to 0xAA55 (i.e., byte 510 contains 0x55 and byte 511 contains 0xAA). |
| **Reserved**              | 512         | Logical Block Size - 512 | The rest of the logical block, if any, is reserved. Set to zero.          |

One of the Partition Records shall be as defined in table 12, reserving the entire space on the disk after the Protective MBR itself for the GPT disk layout.

### Table 5-4 Protective MBR Partition Record protecting the entire disk

<table>
<thead>
<tr>
<th>Mnemonic</th>
<th>Byte Offset</th>
<th>Byte Length</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>BootIndicator</strong></td>
<td>0</td>
<td>1</td>
<td>Set to 0x00 to indicate a non-bootable partition. If set to any value other than 0x00 the behavior of this flag on non-UEFI systems is undefined. Must be ignored by UEFI implementations.</td>
</tr>
<tr>
<td><strong>StartingCHS</strong></td>
<td>1</td>
<td>3</td>
<td>Set to 0x000200, corresponding to the Starting LBA field.</td>
</tr>
</tbody>
</table>
The remaining Partition Records shall each be set to zeros.

**Figure 5-2** shows an example of a GPT disk layout with four partitions with a protective MBR.

**Figure 5-3** shows an example of a GPT disk layout with four partitions with a protective MBR, where the disk capacity exceeds LBA 0xFFFFFFFF.
5.2.4 Partition Information

Install an EFI_PARTITION_INFO protocol on each of the device handles that logical EFI_BLOCK_IO прототипы are installed.

5.3 GUID Partition Table (GPT) Disk Layout

5.3.1 GPT overview

The GPT partitioning scheme is depicted in Figure 5-4. The GPT Header (see Section 5.3.2) includes a signature and a revision number that specifies the format of the data bytes in the partition header. The GUID Partition Table Header contains a header size field that is used in calculating the CRC32 that confirms the integrity of the GPT Header. While the GPT Header’s size may increase in the future it cannot span more than one logical block on the device.

LBA 0 (i.e., the first logical block) contains a protective MBR (see Section 5.2.3).

Two GPT Header structures are stored on the device: the primary and the backup. The primary GPT Header must be located in LBA 1 (i.e., the second logical block), and the backup GPT Header must be located in the last LBA of the device. Within the GPT Header the My LBA field contains the LBA of the GPT Header itself, and the Alternate LBA field contains the LBA of the other GPT Header. For example, the primary GPT Header’s My LBA value would be 1 and its Alternate LBA would be the value for the last LBA of the device. The backup GPT Header’s fields would be reversed.

The GPT Header defines the range of LBAs that are usable by GPT Partition Entries. This range is defined to be inclusive of First Usable LBA through Last Usable LBA on the logical device. All data stored on the volume must be stored between the First Usable LBA through Last Usable LBA, and only the data structures defined by UEFI to manage partitions may reside outside of the usable space. The value of Disk GUID is a GUID that uniquely identifies the entire GPT Header and all its associated storage. This value can be used to uniquely identify the disk. The start of the GPT Partition Entry Array is located at the LBA indicated by the Partition Entry LBA field. The size of a GUID Partition Entry element is defined in the Size Of Partition Entry field. There is a 32-bit CRC of the GPT Partition Entry Array that is stored in the GPT Header in Partition Entry Array CRC32 field.

The size of the GPT Partition Entry Array is Size Of Partition Entry multiplied by Number Of Partition Entries. If the size of the GUID Partition Entry Array is not an even multiple of the logical block size, then any space left over in the last logical block is Reserved and not covered by the Partition Entry Array CRC32 field. When a GUID Partition Entry is updated, the Partition Entry Array CRC32 must be updated. When the Partition Entry Array CRC32 is updated, the GPT Header CRC must also be updated, since the Partition Entry Array CRC32 is stored in the GPT Header.
The primary GPT Partition Entry Array must be located after the primary GPT Header and end before the First Usable LBA. The backup GPT Partition Entry Array must be located after the Last Usable LBA and end before the backup GPT Header.

Therefore the primary and backup GPT Partition Entry Arrays are stored in separate locations on the disk. Each GPT Partition Entry defines a partition that is contained in a range that is within the usable space declared by the GPT Header. Zero or more GPT Partition Entries may be in use in the GPT Partition Entry Array. Each defined partition must not overlap with any other defined partition. If all the fields of a GPT Partition Entry are zero, the entry is not in use. A minimum of 16,384 bytes of space must be reserved for the GPT Partition Entry Array.

If the block size is 512, the First Usable LBA must be greater than or equal to 34 (allowing 1 block for the Protective MBR, 1 block for the Partition Table Header, and 32 blocks for the GPT Partition Entry Array); if the logical block size is 4096, the First Useable LBA must be greater than or equal to 6 (allowing 1 block for the Protective MBR, 1 block for the GPT Header, and 4 blocks for the GPT Partition Entry Array).

The device may present a logical block size that is not 512 bytes long. In ATA, this is called the Long Logical Sector feature set; an ATA device reports support for this feature set in IDENTIFY DEVICE data word 106 bit 12 and reports the number of words (i.e., 2 bytes) per logical sector in IDENTIFY DEVICE data words 117-118 (see ATA8-ACS). A SCSI device reports its logical block size in the READ CAPACITY parameter data Block Length In Bytes field (see SBC-3).

The device may present a logical block size that is smaller than the physical block size (e.g., present a logical block size of 512 bytes but implement a physical block size of 4,096 bytes). In ATA, this is called the Long Physical Sector feature set; an ATA device reports support for this feature set in IDENTIFY DEVICE data word 106 bit 13 and reports the Physical Sector Size/Logical Sector Size exponential ratio in IDENTIFY DEVICE data word 106 bits 3-0 (See ATA8-ACS). A SCSI device reports its logical block size/physical block exponential ratio in the READ CAPACITY (16) parameter data Logical Blocks Per Physical
Block Exponent field (see SBC-3). These fields return $2^x$ logical sectors per physical sector (e.g., 3 means $2^3=8$ logical sectors per physical sector).

A device implementing long physical blocks may present logical blocks that are not aligned to the underlying physical block boundaries. An ATA device reports the alignment of logical blocks within a physical block in IDENTIFY DEVICE data word 209 (see ATA8-ACS). A SCSI device reports its alignment in the READ CAPACITY (16) parameter data Lowest Aligned Logical Block Address field (see SBC-3). Note that the ATA and SCSI fields are defined differently (e.g., to make LBA 63 aligned, ATA returns a value of 1 while SCSI returns a value of 7).

In SCSI devices, the Block Limits VPD page Optimal Transfer Length Granularity field (see SBC-3) may also report a granularity that is important for alignment purposes (e.g., RAID controllers may return their RAID stripe depth in that field).

GPT partitions should be aligned to the larger of:

- the physical block boundary, if any
- the optimal transfer length granularity, if any.

For example:

- If the logical block size is 512 bytes, the physical block size is 4,096 bytes (i.e., 512 bytes x 8 logical blocks), there is no optimal transfer length granularity, and logical block 0 is aligned to a physical block boundary, then each GPT partition should start at an LBA that is a multiple of 8.

- If the logical block size is 512 bytes, the physical block size is 8,192 bytes (i.e., 512 bytes x 16 logical blocks), the optimal transfer length granularity is 65,536 bytes (i.e., 512 bytes x 128 logical blocks), and logical block 0 is aligned to a physical block boundary, then each GPT partition should start at an LBA that is a multiple of 128.

To avoid the need to determine the physical block size and the optimal transfer length granularity, software may align GPT partitions at significantly larger boundaries. For example, assuming logical block 0 is aligned, it may use LBAs that are multiples of 2,048 to align to 1,048,576 byte (1 MiB) boundaries, which supports most common physical block sizes and RAID stripe sizes.

References are as follows:

ISO/IEC 24739-200 [ANSI INCITS 452-2008] AT Attachment 8 - ATA/ATAPI Command Set (ATA8-ACS). By the INCITS T13 technical committee. (See “Links to UEFI-Related Documents” (http://uefi.org/uefi under the headings “InterNational Committee on Information Technology Standards (INCITS)” and “INCITs T13 technical committee”).


5.3.2 GPT Header

Table 5-5 defines the GPT Header.
### Table 5-5 GPT Header

<table>
<thead>
<tr>
<th>Mnemonic</th>
<th>Byte Offset</th>
<th>Byte Length</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Signature</strong></td>
<td>0</td>
<td>8</td>
<td>Identifies EFI-compatible partition table header. This value must contain the ASCII string “EFI PART”, encoded as the 64-bit constant 0x5452415020494645.</td>
</tr>
<tr>
<td><strong>Revision</strong></td>
<td>8</td>
<td>4</td>
<td>The revision number for this header. This revision value is not related to the UEFI Specification version. This header is version 1.0, so the correct value is 0x00010000.</td>
</tr>
<tr>
<td><strong>HeaderSize</strong></td>
<td>12</td>
<td>4</td>
<td>Size in bytes of the GPT Header. The <strong>HeaderSize</strong> must be greater than or equal to 92 and must be less than or equal to the logical block size.</td>
</tr>
<tr>
<td><strong>HeaderCRC32</strong></td>
<td>16</td>
<td>4</td>
<td>CRC32 checksum for the GPT Header structure. This value is computed by setting this field to 0, and computing the 32-bit CRC for <strong>HeaderSize</strong> bytes.</td>
</tr>
<tr>
<td><strong>Reserved</strong></td>
<td>20</td>
<td>4</td>
<td>Must be zero.</td>
</tr>
<tr>
<td><strong>MyLBA</strong></td>
<td>24</td>
<td>8</td>
<td>The LBA that contains this data structure.</td>
</tr>
<tr>
<td><strong>AlternateLBA</strong></td>
<td>32</td>
<td>8</td>
<td>LBA address of the alternate GPT Header.</td>
</tr>
<tr>
<td><strong>FirstUsableLBA</strong></td>
<td>40</td>
<td>8</td>
<td>The first usable logical block that may be used by a partition described by a GUID Partition Entry.</td>
</tr>
<tr>
<td><strong>LastUsableLBA</strong></td>
<td>48</td>
<td>8</td>
<td>The last usable logical block that may be used by a partition described by a GUID Partition Entry.</td>
</tr>
<tr>
<td><strong>DiskGUID</strong></td>
<td>56</td>
<td>16</td>
<td>GUID that can be used to uniquely identify the disk.</td>
</tr>
<tr>
<td><strong>PartitionEntryLBA</strong></td>
<td>72</td>
<td>8</td>
<td>The starting LBA of the GUID Partition Entry array.</td>
</tr>
<tr>
<td><strong>NumberOfPartitionEntries</strong></td>
<td>80</td>
<td>4</td>
<td>The number of Partition Entries in the GUID Partition Entry array.</td>
</tr>
<tr>
<td><strong>SizeOfPartitionEntry</strong></td>
<td>84</td>
<td>4</td>
<td>The size, in bytes, of each the GUID Partition Entry structures in the GUID Partition Entry array. This field shall be set to a value of 128 x 2^n where n is an integer greater than or equal to zero (e.g., 128, 256, 512, etc.). <strong>NOTE:</strong> Previous versions of this specification allowed any multiple of 8.</td>
</tr>
<tr>
<td><strong>PartitionEntryArrayCRC32</strong></td>
<td>88</td>
<td>4</td>
<td>The CRC32 of the GUID Partition Entry array. Starts at <strong>PartitionEntryLBA</strong> and is computed over a byte length of <strong>NumberOfPartitionEntries</strong> * <strong>SizeOfPartitionEntry</strong>.</td>
</tr>
<tr>
<td><strong>Reserved</strong></td>
<td>92</td>
<td>BlockSize – 92</td>
<td>The rest of the block is reserved by UEFI and must be zero.</td>
</tr>
</tbody>
</table>
The following test must be performed to determine if a GPT is valid:

- Check the Signature
- Check the Header CRC
- Check that the MyLBA entry points to the LBA that contains the GUID Partition Table
- Check the CRC of the GUID Partition Entry Array

If the GPT is the primary table, stored at LBA 1:

- Check the AlternateLBA to see if it is a valid GPT

If the primary GPT is corrupt, software must check the last LBA of the device to see if it has a valid GPT Header and point to a valid GPT Partition Entry Array. If it points to a valid GPT Partition Entry Array, then software should restore the primary GPT if allowed by platform policy settings (e.g. a platform may require a user to provide confirmation before restoring the table, or may allow the table to be restored automatically). Software must report whenever it restores a GPT.

Software should ask a user for confirmation before restoring the primary GPT and must report whenever it does modify the media to restore a GPT. If a GPT formatted disk is reformatted to the legacy MBR format by legacy software, the last logical block might not be overwritten and might still contain a stale GPT. If GPT-cognizant software then accesses the disk and honors the stale GPT, it will misinterpret the contents of the disk. Software may detect this scenario if the legacy MBR contains valid partitions rather than a protective MBR (see Section 5.2.1).

Any software that updates the primary GPT must also update the backup GPT. Software may update the GPT Header and GPT Partition Entry Array in any order, since all the CRCs are stored in the GPT Header. Software must update the backup GPT before the primary GPT, so if the size of device has changed (e.g. volume expansion) and the update is interrupted, the backup GPT is in the proper location on the disk.

If the primary GPT is invalid, the backup GPT is used instead and it is located on the last logical block on the disk. If the backup GPT is valid it must be used to restore the primary GPT. If the primary GPT is valid and the backup GPT is invalid software must restore the backup GPT. If both the primary and backup GPTs are corrupted this block device is defined as not having a valid GUID Partition Header.

Both the primary and backup GPTs must be valid before an attempt is made to grow the size of a physical volume. This is due to the GPT recovery scheme depending on locating the backup GPT at the end of the device. A volume may grow in size when disks are added to a RAID device. As soon as the volume size is increased the backup GPT must be moved to the end of the volume and the primary and backup GPT Headers must be updated to reflect the new volume size.

### 5.3.3 GPT Partition Entry Array

The GPT Partition Entry Array contains an array of GPT Partition Entries. Table 5-6 defines the GPT Partition Entry.

<table>
<thead>
<tr>
<th>Mnemonic</th>
<th>Byte Offset</th>
<th>Byte Length</th>
<th>Description</th>
</tr>
</thead>
</table>

---
The `SizeOfPartitionEntry` variable in the GPT Header defines the size of each GUID Partition Entry. Each partition entry contains a **Unique Partition GUID** value that uniquely identifies every partition that will ever be created. Any time a new partition entry is created a new GUID must be generated for that partition, and every partition is guaranteed to have a unique GUID. The partition is defined as all the logical blocks inclusive of the `StartingLBA` and `EndingLBA`.

The `PartitionTypeGUID` field identifies the contents of the partition. This GUID is similar to the OS Type field in the MBR. Each filesystem must publish its unique GUID. The `Attributes` field can be used by utilities to make broad inferences about the usage of a partition and is defined in Table 5-8.

The firmware must add the `PartitionTypeGuid` to the handle of every active GPT partition using `EFI_BOOT_SERVICES.InstallProtocolInterface()`. This will allow drivers and applications, including OS loaders, to easily search for handles that represent EFI System Partitions or vendor specific partition types.

Software that makes copies of GPT-formatted disks and partitions must generate new Disk GUID values in the GPT Headers and new **Unique Partition GUID** values in each GPT Partition Entry. If GPT-cognizant software encounters two disks or partitions with identical GUIDs, results will be indeterminate.

### Table 5-7 Defined GPT Partition Entry - Partition Type GUIDs

<table>
<thead>
<tr>
<th>Description</th>
<th>GUID Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>Unused Entry</td>
<td>00000000-0000-0000-0000-000000000000</td>
</tr>
</tbody>
</table>
OS vendors need to generate their own Partition Type GUIDs to identify their partition types.

### Table 5-8 Defined GPT Partition Entry - Attributes

<table>
<thead>
<tr>
<th>Bits</th>
<th>Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Bit 0</td>
<td>Required Partition</td>
<td>If this bit is set, the partition is required for the platform to function. The owner/creator of the partition indicates that deletion or modification of the contents can result in loss of platform features or failure for the platform to boot or operate. The system cannot function normally if this partition is removed, and it should be considered part of the hardware of the system. Actions such as running diagnostics, system recovery, or even OS install or boot could potentially stop working if this partition is removed. Unless OS software or firmware recognizes this partition, it should never be removed or modified as the UEFI firmware or platform hardware may become non-functional.</td>
</tr>
<tr>
<td>Bit 1</td>
<td>No Block IO Protocol</td>
<td>If this bit is set, then firmware must not produce an EFI_BLOCK_IO_PROTOCOL device for this partition. See Section 13.3.2 for more details. By not producing an EFI_BLOCK_IO_PROTOCOL partition, file system mappings will not be created for this partition in UEFI.</td>
</tr>
<tr>
<td>Bit 2</td>
<td>Legacy BIOS Bootable</td>
<td>This bit is set aside by this specification to let systems with traditional PC-AT BIOS firmware implementations inform certain limited, special-purpose software running on these systems that a GPT partition may be bootable. For systems with firmware implementations conformance to this specification, the UEFI boot manager (see chapter 3) must ignore this bit when selecting a UEFI-compliant application, e.g., an OS loader (see 2.1.3). Therefore there is no need for this specification to define the exact meaning of this bit.</td>
</tr>
<tr>
<td>Bits 3-47</td>
<td>Undefined and must be zero. Reserved for expansion by future versions of the UEFI specification.</td>
<td></td>
</tr>
<tr>
<td>Bits 48-63</td>
<td>Reserved for GUID specific use. The use of these bits will vary depending on the PartitionTypeGUID. Only the owner of the PartitionTypeGUID is allowed to modify these bits. They must be preserved if Bits 0–47 are modified.</td>
<td></td>
</tr>
</tbody>
</table>

**Related Definitions:**

```c
#pragma pack(1)
///<
/// GPT Partition Entry.
///<
typedef struct {
    EFI_GUID PartitionTypeGUID;
    EFI_GUID UniquePartitionGUID;
    EFI_LBA StartingLBA;
    EFI_LBA EndingLBA;
    UINT64 Attributes;
    CHAR16 PartitionName[36];
};
```
6 - Block Translation Table (BTT) Layout

This specification defines the Block Translation Table (BTT) metadata layout. The following sub-sections outline the BTT format that is utilized on the media, the data structures involved, and a detailed description of how SW is to interpret the BTT layout.

6.1 Block Translation Table (BTT) Background

A namespace defines a contiguously-addressed range of Non-Volatile Memory conceptually similar to a SCSI Logical Unit (LUN) or a NVM Express namespace.

Any namespace being utilized for block storage may contain a Block Translation Table (BTT), which is a layout and set of rules for doing block I/O that provide powerfail write atomicity of a single block. Traditional block storage, including hard disks and SSDs, usually protect against torn sectors, which are sectors partially written when interrupted by power failure. Existing software, mostly file systems, depend on this behavior, often without the authors realizing it. To enable such software to work correctly on namespaces supporting block storage access, the BTT layout defined by this document sub-divides a namespace into one or more BTT Arenas, which are large sections of the namespace that contain the metadata required to provide the desired write atomicity. Each of these BTT Arenas contains a metadata layout as shown in Figure 6-1 and Figure 6-2.

![Image of BTT Arena layout]

Figure 6-1 The BTT Layout in a BTT Arena

Each arena contains the layout shown in Figure: The BTT Layout in a BTT Arena, the primary info block, data area, map, flog, and a backup info block. Each of these areas is described in the following sections. When the namespace is larger than 512 GiB, multiple arenas are required by the BTT layout, as shown in Figure 6-2. Each namespace using a BTT is divided into as many 512 GiB arenas as shall fit, followed by a smaller arena to contain any remaining space as appropriate. The smallest arena size is 16MiB so the last arena size shall be between 16MiB and 512GiBs. Any remaining space less than 16MiB is unused. Because...
of these rules for arena placement, software can locate every primary Info block and every backup Info block without reading any metadata, based solely on the namespace size.

![Figure 6-2 A BTT With Multiple Arenas in a Large Namespace](image)

### 6.2 Block Translation Table (BTT) Data Structures

The following sub-sections outline the data structures associated with the BTT Layout.

#### 6.2.1 BTT Info Block

```
// Alignment of all BTT structures
#define EFI_BTT_ALIGNMENT 4096
#define EFI_BTT_INFO_UNUSED_LEN 3968
#define EFI_BTT_INFO_BLOCK_SIG_LEN 16

// Constants for Flags field
#define EFI_BTT_INFO_BLOCK_FLAGS_ERROR 0x00000001

// Constants for Major and Minor version fields
#define EFI_BTT_INFO_BLOCK_MAJOR_VERSION 2
#define EFI_BTT_INFO_BLOCK_MINOR_VERSION 0

typedef struct _EFI_BTT_INFO_BLOCK {   
  CHAR8  Sig[EFI_BTT_INFO_BLOCK_SIG_LEN];
  EFI_GUID  Uuid;
  EFI_GUID  ParentUuid;
  UINT32  Flags;
  UINT16  Major;
  UINT16  Minor;
  UINT32  ExternalLbaSize;
```
UINT32 ExternalNLba;
UINT32 InternallLbaSize;
UINT32 InternalNLba;
UINT32 NFree;
UINT32 InfoSize;
UINT64 NextOff;
UINT64 DataOff;
UINT64 MapOff;
UINT64 FlogOff;
UINT64 InfoOff;
CHAR8 Unused[EFI_BTT_INFO_UNUSED_LEN];
UINT64 Checksum;
} EFI_BTT_INFO_BLOCK

Sig
Signature of the BTT Index Block data structure. Shall be “BTT_ARENA_INFO\0\0”.

Uuid
UUID identifying this BTT instance. A new UUID is created each time the initial BTT Arenas are written. This value shall be identical across all BTT Info Blocks within all arenas within a namespace.

ParentUuid
UUID of containing namespace, used when validating the BTT Info Block to ensure this instance of the BTT layout is intended for the current surrounding namespace, and not left over from a previous namespace that used the same area of the media. This value shall be identical across all BTT Info Blocks within all arenas within a namespace.

Flags
Boolean attributes of this BTT Info Block. See the additional description below on the use of the flags. The following values are defined:

EFI_BTT_INFO_BLOCK_FLAGS_ERROR – The BTT Arena is in the error state. When a BTT implementation discovers issues such as inconsistent metadata or lost metadata due to unrecoverable media errors, the error bit for the associated arena shall be set. See the BTT Theory of Operation section regarding handling of EFI_BTT_INFO_BLOCK_FLAGS_ERROR.

Major
Major version number. Currently at version 2. This value shall be identical across all BTT Info Blocks within all arenas within a namespace.

Minor
Minor version number. Currently at version 0. This value shall be identical across all BTT Info Blocks within all arenas within a namespace.
ExternalLbaSize
Advertised LBA size in bytes. I/O requests shall be in this size chunk. This value shall be identical across all BTT Info Blocks within all arenas within a namespace.

ExternalNLba
Advertised number of LBAs in this arena. The sum of this field, across all BTT Arenas, is the total number of available LBAs in the namespace.

InternalLbaSize
Internal LBA size shall be greater than or equal to ExternalLbaSize and shall not be smaller than 512 bytes. Each block in the arena data area is this size in bytes and contains exactly one block of data. Optionally, this may be larger than the ExternalLbaSize due to alignment padding between LBAs. This value shall be identical across all BTT Info Blocks within all arenas within a namespace.

InternalNLba
Number of internal blocks in the arena data area. This shall be equal to ExternalNLba + NFree because each internal lba is either mapped to an external lba or shown as free in the flog.

NFree
Number of free blocks maintained for writes to this arena. NFree shall be equal to InternalNLba – ExternalNLba. This value shall be identical across all BTT Info Blocks within all arenas within a namespace.

InfoSize
The size of this info block in bytes. This value shall be identical across all BTT Info Blocks within all arenas within a namespace.

NextOff
Offset of next arena, relative to the beginning of this arena. An offset of 0 indicates that no arenas follow the current arena. This field is provided for convenience as the start of each arena can be calculated from the size of the namespace as described in the Theory of Operation – Validating BTT Arenas at start-up description. This value shall be identical in the primary and backup BTT Info Blocks within an arena.

DataOff
Offset of the data area for this arena, relative to the beginning of this arena. The internal-LBA number zero lives at this offset. This value shall be identical in the primary and backup BTT Info Blocks within an arena.

MapOff
Offset of the map for this arena, relative to the beginning of this arena. This value shall be identical in the primary and backup BTT Info Blocks within an arena.

FlogOff
Offset of the flog for this arena, relative to the beginning of this arena. This value shall be identical in the primary and backup BTT Info Blocks within an arena.
**InfoOff**

Offset of the backup copy of this arena’s info block, relative to the beginning of this arena. This value shall be identical in the primary and backup BTT Info Blocks within an arena.

**Reserved**

Shall be zero.

**Checksum**

64-bit Fletcher64 checksum of all fields. This field is considered as containing zero when the checksum is computed.

**BTT Info Block Description**

The existence of a valid BTT Info Block is used to determine whether a namespace is used as a BTT block device.

Each BTT Arena contains two BTT Info Blocks, a primary copy at the beginning of the BTT Arena, at address offset 0, and ends with an identical backup BTT Info Block, in the highest block available in the arena aligned on a EFI_BTT_ALIGNMENT boundary. When writing the BTT layout, implementations shall write out the info blocks from the highest arena to the lowest, writing the backup info block and other BTT data structures before writing the primary info block. Writing the layout in this manner shall ensure that a valid BTT layout is only detected after the entire layout has been written.

### 6.2.2 BTT Map Entry

```c
typedef struct _EFI_BTT_MAP_ENTRY {
    UINT32 PostMapLba : 30;
    UINT32 Error : 1;
    UINT32 Zero : 1;
} EFI_BTT_MAP_ENTRY;
```

**PostMapLba**

Post-map LBA number (block number in this arena’s data area)

**Error**

When set and **Zero** is not set, reads on this block return an error. Writes to this block clear this flag.

**Zero**

When set and **Error** is not set, reads on this block return a full block of zeros. Writes to this block clear this flag.

**BTT Map Description**

The BTT Map area maps an LBA that indexes into the arena, to its actual location. The BTT Map is located as high as possible in the arena, after room for the backup info block and flog (and any required alignment) has been taken into account. The terminology pre-map LBA and post-map LBA is used to describe the input and output values of this mapping.

The BTT Map area is indexed by the pre-map LBA and each entry in the map contains the 30 bit post-map LBA and bits to indicate if there is an error or if LBA contains zeroes (see EFI_BTT_MAP_ENTRY).
The **Error** and **Zero** bits indicate conditions that cannot both be true at the same time, so that combination is used to indicate a *normal* map entry, where no error or zeroed block is indicated. The error condition is indicated only when the **Error** bit is set and the **Zero** bit is clear, with similar logic for the zero block condition. When neither condition is indicated, both **Error** and **Zero** are set to indicate a map entry in its normal, non-error state. This leaves the case where both **Error** and **Zero** are bits are zero, which is the initial state of all map entries when the BTT layout is first written. Both bits zero means that the map entry contains the initial *identity* mapping where the pre-map LBA is mapped to the same post-map LBA. Defining the map this way allows an implementation to leverage the case where the initial contents of the namespace is known to be zero, requiring no writes to the map when writing the layout. This can greatly improve the layout time since the map is the largest BTT data structure written during layout.

### 6.2.3 BTT Flog

```c
// Alignment of each flog structure
#define EFI_BTT_FLOG_ENTRY_ALIGNMENT 64

typedef struct _EFI_BTT_FLOG {
  UINT32 Lba0;
  UINT32 OldMap0;
  UINT32 NewMap0;
  UINT32 Seq0;
  UINT32 Lba1;
  UINT32 OldMap1;
  UINT32 NewMap1;
  UINT32 Seq1;
} EFI_BTT_FLOG
```

**Lba0**

Last pre-map LBA written using this flog entry. This value is used as an index into the BTT Map when updating it to complete the transaction.

**OldMap0**

Old post-map LBA. This is the old entry in the map when the last write using this flog entry occurred. If the transaction is complete, this LBA is now the free block associated with this flog entry.

**NewMap0**

New post-map LBA. This is the block allocated when the last write using this flog entry occurred. By definition, a write transaction is complete if the BTT Map entry contains this value.

**Seq0**

The **Seq0** field in each flog entry is used to determine which set of fields is newer between the two sets (Lba0, OldMap0, NewMpa0, Seq0 vs Lba1, Oldmap1, NewMap1, Seq1). Updates to a flog entry shall always be made to the older set of fields and shall be implemented carefully so that the **Seq0** bits are only written after the other fields are known to be committed to persistence. The figure below shows the progression of the **Seq0** bits over time, where the newer entry is indicated by a value that is clockwise of the older value.
Figure 6-3 Cyclic Sequence Numbers for Flog Entries

**Lba1**
Alternate lba entry

**OldMap1**
Alternate old entry

**NewMap1**
Alternate new entry

**Seq1**
Alternate Seq entry

**BTT Flog Description**
The BTT Flog is so named to illustrate that it is both a free list and a log, rolled into one data structure. The flog size is determined by the NFree field in the BTT Info Block which determines how many of these flog entries there are. The flog location is the highest address in the arena after space for the backup info block and alignment requirements have been taken in account.

**6.2.4 BTT Data Area**
Starting from the low address to high, the BTT Data Area starts immediately after the BTT Info Block and extends to the beginning of the BTT Map data structure. The number of internal data blocks that can be stored in an arena is calculated by first calculating the necessary space required for the BTT Info Blocks, map, and flog (plus any alignment required), subtracting that amount from the total arena size, and then calculating how many blocks fit into the resulting space.

**6.2.5 NVDIMM Label Protocol Address Abstraction Guid**
This version of the BTT layout and behavior is collectively described by the AddressAbstractionGuid in the UEFI NVDIMM Label protocol section utilizing this GUID:
#define EFI_BTT_ABSTRACTION_GUID \ 
  {0x18633bfc,0x1735,0x4217, \ 
  {0x8a,0xc9,0x17,0x23,0x92,0x82,0xd3,0xf8}

6.3 BTT Theory of Operation

This section outlines the theory of operation for the BTT and describes the responsibilities that any software implementation shall follow.

A specific instance of the BTT layout depends on the size of the namespace and three administrative choices made at the time the initial layout is created:

- **ExternalLbaSize**: the desired block size
- **InternalLbaSize**: the block size with any internal padding
- **NFree**: the number of concurrent writes supported by the layout

The BTT data structures do not support an **InternalLbaSize** smaller than 512 bytes, so if **ExternalLbaSize** is smaller than 512 bytes, the **InternalLbaSize** shall be rounded up to 512. For performance, the **InternalLbaSize** may also include some padding bytes. For example, a BTT layout supporting 520-byte blocks may use 576-byte blocks internally in order to round up the size to a multiple of a 64-byte cache line size. In this example, the **ExternalLbaSize**, visible to software above the BTT software, would be 520 bytes, but the **InternalLbaSize** would be 576 bytes.

Once these administrative choices above are determined, the namespace is divided up into **arenas**, as described in the **BTT Arenas** section, where each arena uses the same values for **ExternalLbaSize**, **InternalLbaSize**, and **Nfree**.

### 6.3.1 BTT Arenas

In order to reduce the size of BTT metadata and increase the possibility of concurrent updates, the BTT layout in a namespace is divided into **arenas**. An arena cannot be larger than 512GiB or smaller than 16MiB. A namespace is divided into as many 512GiB arenas that shall fit, starting from offset zero and packed together without padding, followed by one arena smaller than 512GiB if the remaining space is at least 16MiB. The smaller area size is rounded down to be a multiple of EFI_BTT_ALIGNMENT if necessary.

Because of these rules, the location and size of every BTT Arena in a namespace can be determined from the namespace size.

Within an arena, the amount of space used for the Flog is **NFree** times the amount of space required for each Flog entry. Flog entries shall be aligned on 64-byte boundaries. In addition, the full BTT Flog table shall be aligned on a EFI_BTT_ALIGNMENT boundary and have a size that is padded to be multiple of EFI_BTT_ALIGNMENT. In summary, the space in an arena taken by the Flog is:

\[
FlogSize = \text{roundup}(NFree \times \text{roundup}(\text{sizeof}(\text{EFI_BTT_FLOG}), \text{EFI_BTT_FLOG_ENTRY_ALIGNMENT}), \text{EFI_BTT_ALIGNMENT})
\]

Within an arena, the amount of space available for data blocks and the associated Map is the arena size minus the space used for the BTT Info Blocks and the Flog:
\[ \text{DataAndMapSize} = \text{ArenaSize} - 2 \times \text{sizeof(EFI_BTT_INFO_BLOCK)} - \text{FlogSize} \]

Within an arena, the number of data blocks is calculated by dividing the available space, DataAndMapSize, by the InternalLbaSize plus the map overhead required for each block, and rounding down the result to ensure the data area is aligned on a EFI_BTT_ALIGNMENT boundary:

\[ \text{InternalNLba} = \frac{\text{DataAndMapSize} - \text{EFI_BTT_ALIGNMENT}}{\text{InternalLbaSize} + \text{sizeof(EFI_BTT_MAP_ENTRY)}} \]

With the InternalNLba value known, the calculation for the number of external LBAs subtracts off NFree for the pool of unadvertised free blocks:

\[ \text{ExternalNLba} = \text{InternalNLba} - \text{Nfree} \]

Within an arena, the number of bytes required for the BTT Map is one entry for each external LBA, plus any alignment required to maintain an alignment of EFI_BTT_ALIGNMENT for the entire map:

\[ \text{MapSize} = \text{roundup}(\text{ExternalNLba} \times \text{sizeof(EFI_BTT_MAP_ENTRY)}, \text{EFI_BTT_ALIGNMENT}) \]

The number of concurrent writes allowed for an arena is based on the NFree value chosen at BTT layout time. For example, choosing NFree of 256 means the BTT Arena shall have 256 free blocks to use for in-flight write operations. Since BTT Arenas each have NFree free blocks, the number of concurrent writes allowed in a namespace may be larger when there are multiple arenas and the writes are spread out between multiple arenas.

### 6.3.2 Atomicity of Data Blocks in an Arena

The primary reason for the BTT is to provide failure atomicity when writing data blocks, so that any write of a single block cannot be torn by interruptions such as power loss. The BTT provides this by maintaining a pool of free blocks which are not part of the capacity advertised to software layers above the BTT software. The BTT Data Area is large enough to hold the advertised capacity as well as the pool of free blocks. The BTT software manages the blocks in the BTT Data Area as a list of internal LBAs, which are block numbers only visible internally to the BTT software. The block numbers that make up the advertised capacity are known as external LBAs, and at any given point in time, each one of those external LBAs is mapped by the BTT Map to one of the blocks in the BTT Data Area. Each block write done by the BTT software starts by allocating one of the free blocks, writing the data to it, and only when that block is fully persistent (including any flushes required), are steps taken to make that block active, as outlined in the BTT Theory of Operations – Write Path section.

The BTT Flog (a combination of a free list and a log) is at the heart of the atomic updates when writing blocks. The “quiet” state of a BTT Flog, when no in-flight writes are happening and no recovery steps are outstanding, is that the NFree free blocks currently available for writes are contained in the OldMap fields in the Flog entries. A write shall use one of those Flog entries to find a free block to write to, and then the Lba and NewMap fields in the Flog are used as a write-ahead-log for the BTT Map update when the data portion of the write is complete, as described in the Validating the Flog at start-up section.

It is up to run-time logic in the BTT software to ensure that only one Flog entry is in use at a time, and that any reads still executing on the block indicated by the OldMap entry have finished before starting a write using that block.
6.3.3 Atomicity of BTT Data Structures

Byte-addressable persistent media may not support atomic updates larger than 8-bytes, so any data structure larger than 8-bytes in the BTT uses software-implemented atomicity for updates. Note that 8-byte write atomicity, meaning an 8-byte store to the persistent media cannot be torn by interruptions such as power failures, is a minimal requirement for using the BTT described in this document.

There are four types of data structures in the BTT:

- The BTT Info Blocks
- The BTT Map
- The BTT Flog
- The BTT Data Area

The BTT Map entries are 4-bytes in size, and so can be updated atomically with a single store instruction. All other data structures are updated by following the rules described in this document, which update an inactive version of the data structure first, followed by steps to make it active atomically.

For the BTT Info Blocks, atomicity is provided by always writing the backup Info block first, and only after that update is fully persistent (the block checksums correctly), is the primary BTT Info Block updated as described in the Writing the initial BTT layout section. Recovery from an interrupted update is provided by checking the primary Info block’s checksum on start-up, and if it is bad, copying the backup Info block to the primary to complete the interrupted update as described in the Validating BTT Arenas at start-up section.

For the BTT Flog, each entry is double-sized, with two complete copies of every field (Lba, OldMap, NewMap, Seq). The active entry has the higher Seq number, so updates always write to the inactive fields, and once those fields are fully persistent, the Seq field for the inactive entry is updated to make it become the active entry atomically. This is described in the Validating the Flog at start-up section.

For the BTT Data Area, all block writes can be thought of as allocating writes, where an inactive block is chosen from the free list maintained by the Flog, and only after the new data written to that block is fully persistent, that block is made active atomically by updating the Flog and Map entries as described in the Write Path section.

6.3.4 Writing the Initial BTT layout

The overall layout of the BTT relies on the fact that all arenas shall be 512GiB in size, except the last arena which is a minimum of 16MiB. Initializing the BTT on-media structures only happens once in the lifetime of a BTT, when it is created. This sequence assumes that software has determined that new BTT layout needs to be created and the total raw size of the namespace is known.

Immediately before creating a new BTT layout, the UUID of the surrounding namespace may be updated to a newly-generated UUID. This optional step, depending on the needs of a BTT software implementation, has the effect of invalidating any previous BTT Info Blocks in the namespace and ensuring the detection of an invalid layout if the BTT layout creation process is interrupted. This detection works because the parent UUID field

The on-media structures in the BTT layout may be written out in any order except for the BTT Info Blocks, which shall be written out as the last step of the layout, starting from the last arena (highest offset in the namespace) to the first arena (lowest offset in the namespace), writing the backup BTT Info Block in each
arena first, then writing the primary BTT Info block for that arena second. This allows the detection of an incomplete BTT layout when the algorithm in the Validating BTT Arenas at start-up section is executed.

Since the number of internal LBAs for an arena exceeds the number of external LBAs by \textit{NFree}, there are enough internal LBA numbers to fully initialize the BTT Map as well as the BTT Flog, where the BTT Flog is initialized with the \textit{NFree} highest internal LBA numbers, and the rest are used in the BTT Map.

The BTT Map in each arena is initialized to zeros. Zero entries in the map indicate the identity mapping of all pre-map LBAs to the corresponding post-map LBAs. This uses all but \textit{NFree} of the internal LBAs, leaving \textit{Nfree} of them for the BTT Flog.

The BTT Flog in each arena is initialized by starting with all zeros for the entire flog area, setting the \textit{Lba0} field in each flog entry to unique pre-map LBAs, zero through \textit{NFree} – 1, and both \textit{OldMap0} and \textit{NewMap0} fields in each flog entry are set to one of the remaining internal LBAs. For example, flog entry zero would have \textit{Lba0} set to 0, and \textit{OldMap0} and \textit{NewMap0} both set to the first internal LBA not represented in the map (since there are \textit{ExternalNLba} entries in the map, the next available internal LBA is equal to \textit{ExternalNLba}).

6.3.5 Validating BTT Arenas at start-up

When software prepares to access the BTT layout in a namespace, the first step is to check the BTT Arenas for consistency. Reading and validating BTT Arenas relies on the fact that all arenas shall be 512GiB in size, except the last arena which is a minimum of 16MiB.

The following tests shall pass before software considers the BTT layout to be valid:

- For each BTT Arena:
  - \textit{ReadAndVerifyPrimaryBttInfoBlock}
    - If the read of the primary BTT Info Block fails, goto \textit{ReadAndVerifyBackupBttInfoBlock}
    - If the primary BTT Info Block contains an incorrect \textit{Sig} field it is invalid, goto \textit{ReadAndVerifyBackupBttInfoBlock}
    - If the primary BTT Info Block ParentUuid field does not match the UUID of the surrounding namespace, goto \textit{ReadAndVerifyBackupBttInfoBlock}
    - If the primary BTT Info Block contains an incorrect \textit{Checksum} it is invalid, goto \textit{ReadAndVerifyBackupBttInfoBlock}
    - The primary BTT Info Block is valid. Use the \textit{NextOff} field to find the start of the next arena and continue BTT Info Block validation, goto \textit{ReadAndVerifyPrimaryBttInfoBlock}
  - \textit{ReadAndVerifyBackupBttInfoBlock}
    - Determine the location of the backup BTT Info Block:
      1. All of the arenas shall be the full 512GiB data area size except the last arena which is at least 16MiB.
      2. The backup BTT Info Block is the last EFI\_BTT\_ALIGNMENT aligned block in the arena.
    - If the read of the backup BTT Info Block at the high address of the BTT Arena fails, neither copy could be read, and software shall assume that there is no valid BTT metadata layout for the namespace.
• If the backup BTT Info Block contains an incorrect **Sig** field it is invalid, and software shall assume that there is no valid BTT metadata layout for the namespace

• If the backup BTT Info Block ParentUuid field does not match the UUID of the surrounding namespace it is invalid, and software shall assume that there is no valid BTT metadata layout for the namespace

• If the backup BTT Info Block contains an incorrect **Checksum** it is invalid, and software shall assume that there is no valid BTT metadata layout for the namespace

• The backup BTT Info Block is valid. Since the primary copy is bad, software shall copy the contents of the valid backup BTT Info Block down to the primary BTT Info Block before validation of all of the BTT Info Blocks in all of the arenas can complete successfully.

### 6.3.6 Validating the Flog entries at start-up

After validating the BTT Info Blocks as described in the **Validating BTT Arenas at start-up** section, the next step software shall take is to validate the BTT Flog entries. When blocks of data are being written, as described in the **Write Path** section below, the persistent Flog and Map states are not updated until the free block is written with new data. This ensures a power failure at any point during the data transfer is harmless, leaving the partially written data in a free block that remains free. Once the Flog is updated (made atomic by the Seq bits in the Flog entry), the write algorithm is committed to the update and a power failure from this point in the write flow onwards shall be handled by completing the update to the BTT Map on recovery. The Flog contains all the information required to complete the Map entry update.

Note that the Flog entry recovery outlined here is intended to happen single-threaded, on an **inactive** BTT (before the BTT block namespace is allowed to accept I/O requests). The maximum amount of time required for recovery is determined by **NFree**, but is only a few loads and a single store (and the corresponding cache flushes) for each incomplete write discovered.

The following steps are executed for each flog entry in each arena, to recover any interrupted writes and to verify the flog entries are consistent at start up. Any consistency issues found during these steps results in setting the error state (**EFI_BTT_INFO_BLOCK_FLAGS_ERROR**) for the arena and terminates the flog validation process for this arena.

1. The **Seq0** and **Seq1** fields are examined for the flog entry. If both fields are zero, or both fields are equal to each other, the flog entry is inconsistent. Otherwise, the higher Seq field indicates which set of flog fields to use for the next steps (**Lba0**, **OldMap0**, **NewMap0**, versus **Lba1**, **OldMap1**, **NewMap1**). From this point on in this section, the chosen fields are referenced as Lba, OldMap, and NewMap.

2. If OldMap and NewMap are equal, this is a flog entry that was never used since the initial layout of the BTT was created.

3. The Lba field is checked to ensure it is a valid pre-map LBA (in the range zero to **ExternalNLba** – 1). If the check fails, the flog entry is inconsistent.

4. The BTT Map entry corresponding to the Flog entry Lba field is fetched. Since the Map can contain special zero entries to indicate identity mappings, the fetched entry is adjusted to the corresponding internal LBA when a zero is encountered (by interpreting the entry as the same LBA as the Flog entry Lba field).

5. If the adjusted map entry from the previous step does not match the NewMap field in the Flog entry, and it matches the OldMap field, then an interrupted BTT Map update has been
detected. The recovery step is to write the NewMap field to the BTT Map entry indexed by the Flog entry Lba field.

6.3.7 Read Path

The following high level sequence describes the steps to read a single block of data while utilizing the BTT as is illustrated in the Figure: BTT Read Path Overview below:

1. If EFI_BTT_INFO_BLOCK_FLAGS_ERROR is set in the arena’s BTT Info Block, the BTT software may return an error for the read, or an implementation may choose to continue to provide read-only access and continue these steps.

2. Use the external LBA provided with the read operation to determine which BTT Arena to access. Starting from the first arena (lowest offset in the namespace), and looping through the arena in order, the ExternalNLba field in the BTT Info Block describes how many external LBAs are in that area. Once the correct arena is identified, the external LBAs contained in the lower, skipped, arenas are subtracted from the provided LBA to obtain the pre-map LBA for the selected arena.

3. Use the pre-map LBA to index into the arena’s BTT Map and fetch the map entry.

4. If both the Zero and Error bits are set in the map entry, this indicates a normal entry. The PostMapLba field in the Map entry is used to index into the arena Data Area by multiplying it by the InternalLbaSize and adding the result to the DataOff field from the arena’s BTT Info Block. This provides the location of the data in the arena and software then copies ExternalLbaSize bytes into the provided buffer to satisfy the read request.

5. Otherwise, if only the Error bit is set in the map entry, a read error is returned.

6. Otherwise, if only the Zero bit is set in the map entry, a block of ExternalLbaSize bytes of zeros is copied into the provided buffer to satisfy the read request.

7. Finally, if both Zero and Error bits are clear, this indicates the initial identity mapping and the pre-map LBA is used to index into the arena Data Area by multiplying it by the InternalLbaSize and adding the result to the DataOff field from the arena’s BTT Info Block. This provides the location of the data in the arena and software then copies ExternalLbaSize bytes into the provided buffer to satisfy the read request.
6.3.8 Write Path

The following high level sequence describes the steps to write a single block of data while utilizing the BTT as is illustrated in the Figure: BTT Write Path Overview below:

1. If EFI_BTT_INFO_BLOCK_FLAGS_ERROR is set in the arena’s BTT Info Block, the BTT software shall return an error for the write.

2. Use the external LBA provided with the write operation to determine which BTT Arena to access. Starting from the first arena (lowest offset in the namespace), and looping through the arena in order, the ExternalNLba field in the BTT Info Block describes how many external LBAs are in that area. Once the correct arena is identified, the external LBAs contained in the lower, skipped, arenas are subtracted from the provided LBA to obtain the pre-map LBA for the selected arena.

3. The BTT software allocates one of the Flog entries in the arena to be used for this write. The Flog entry shall not be shared by multiple concurrent writes. The exact method for managing the exclusive use of the Flog entries is BTT software implementation-dependent. There’s no on-media indication of whether a Flog entry is currently allocated to a write request or not. Note that the free block tracked by the Flog entry in the OldMap field, may still have reads from relatively slow threads operating on it. The BTT software implementation shall ensure any such reads have completed before moving to the next step.

4. Lock out access to the BTT Map area associated with the pre-map LBA for the next three steps. The granularity of the locking is implementation-dependent; an implementation may choose to lock individual Map entries, lock the entire BTT Map, or something in-between.

5. Use the pre-map LBA to index into the arena’s BTT Map and fetch the old map entry.

6. Update the Flog entry by writing the inactive set of Flog fields (the lower Seq number). First, update the Lba, OldMap, and NewMap fields with the pre-map LBA, old Map entry, and the
free block chosen above, respectively. Once those fields are fully persistent (with any required flushes completed), the Seq field is updated to make the new fields active. This update of the Seq field commits the write – before this update, the write shall not take place if the operation is interrupted. After the Seq field is updated, the write shall take place even if the operation is interrupted because the Map update in the next step shall take place during the BTT recovery that happens on start-up.

7. Update the Map entry with the free block chosen above.
8. Drop the map lock acquired in step 4 above. The write request is now satisfied.

![Figure 6-5 BTT Write Path Overview](image-url)
7 - Services — Boot Services

This section discusses the fundamental boot services that are present in a UEFI compliant system. The services are defined by interface functions that may be used by code running in the UEFI environment. Such code may include protocols that manage device access or extend platform capability, as well as applications running in the preboot environment, and OS loaders.

Two types of services apply in a compliant system:

- **Boot Services**: Functions that are available before a successful call to `EFI_BOOT_SERVICES.ExitBootServices()`. These functions are described in this section.

- **Runtime Services**: Functions that are available before and after any call to `ExitBootServices()`. These functions are described in Section 8.

During boot, system resources are owned by the firmware and are controlled through boot services interface functions. These functions can be characterized as “global” or “handle-based.” The term “global” simply means that a function accesses system services and is available on all platforms (since all platforms support all system services). The term “handle-based” means that the function accesses a specific device or device functionality and may not be available on some platforms (since some devices are not available on some platforms). Protocols are created dynamically. This section discusses the “global” functions and runtime functions; subsequent sections discuss the “handle-based.”

UEFI applications (including UEFI OS loaders) must use boot services functions to access devices and allocate memory. On entry, an Image is provided a pointer to a system table which contains the Boot Services dispatch table and the default handles for accessing the console. All boot services functionality is available until a UEFI OS loader loads enough of its own environment to take control of the system’s continued operation and then terminates boot services with a call to `ExitBootServices()`.

In principle, the `ExitBootServices()` call is intended for use by the operating system to indicate that its loader is ready to assume control of the platform and all platform resource management. Thus boot services are available up to this point to assist the UEFI OS loader in preparing to boot the operating system. Once the UEFI OS loader takes control of the system and completes the operating system boot process, only runtime services may be called. Code other than the UEFI OS loader, however, may or may not choose to call `ExitBootServices()`. This choice may in part depend upon whether or not such code is designed to make continued use of boot services or the boot services environment.

The rest of this section discusses individual functions. Global boot services functions fall into these categories:

- Event, Timer, and Task Priority Services ([Section 7.1](#))
- Memory Allocation Services ([Section 7.2](#))
- Protocol Handler Services ([Section 7.3](#))
- Image Services ([Section 7.4](#))
- Miscellaneous Services ([Section 7.5](#))
7.1 Event, Timer, and Task Priority Services

The functions that make up the Event, Timer, and Task Priority Services are used during preboot to create, close, signal, and wait for events; to set timers; and to raise and restore task priority levels. See Table 7-1.

Table 7-1 Event, Timer, and Task Priority Functions

<table>
<thead>
<tr>
<th>Name</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>CreateEvent</td>
<td>Boot</td>
<td>Creates a general-purpose event structure</td>
</tr>
<tr>
<td>CreateEventEx</td>
<td>Boot</td>
<td>Creates an event structure as part of an event group</td>
</tr>
<tr>
<td>CloseEvent</td>
<td>Boot</td>
<td>Closes and frees an event structure</td>
</tr>
<tr>
<td>SignalEvent</td>
<td>Boot</td>
<td>Signals an event</td>
</tr>
<tr>
<td>WaitForEvent</td>
<td>Boot</td>
<td>Stops execution until an event is signaled</td>
</tr>
<tr>
<td>CheckEvent</td>
<td>Boot</td>
<td>Checks whether an event is in the signaled state</td>
</tr>
<tr>
<td>SetTimer</td>
<td>Boot</td>
<td>Sets an event to be signaled at a particular time</td>
</tr>
<tr>
<td>RaiseTPL</td>
<td>Boot</td>
<td>Raises the task priority level</td>
</tr>
<tr>
<td>RestoreTPL</td>
<td>Boot</td>
<td>Restores/lowers the task priority level</td>
</tr>
</tbody>
</table>

Execution in the boot services environment occurs at different task priority levels, or TPLs. The boot services environment exposes only three of these levels to UEFI applications and drivers:

- **TPL_APPLICATION**, the lowest priority level
- **TPL_CALLBACK**, an intermediate priority level
- **TPL_NOTIFY**, the highest priority level

Tasks that execute at a higher priority level may interrupt tasks that execute at a lower priority level. For example, tasks that run at the **TPL_NOTIFY** level may interrupt tasks that run at the **TPL_APPLICATION** or **TPL_CALLBACK** level. While **TPL_NOTIFY** is the highest level exposed to the boot services applications, the firmware may have higher task priority items it deals with. For example, the firmware may have to deal with tasks of higher priority like timer ticks and internal devices. Consequently, there is a fourth TPL, **TPL_HIGH_LEVEL**, designed for use exclusively by the firmware.

The intended usage of the priority levels is shown in Table 7-2 from the lowest level (TPL_APPLICATION) to the highest level (TPL_HIGH_LEVEL). As the level increases, the duration of the code and the amount of blocking allowed decrease. Execution generally occurs at the **TPL_APPLICATION** level. Execution occurs at other levels as a direct result of the triggering of an event notification function (this is typically caused by the signaling of an event). During timer interrupts, firmware signals timer events when an event’s “trigger time” has expired. This allows event notification functions to interrupt lower priority code to check devices (for example). The notification function can signal other events as required. After all pending event notification functions execute, execution continues at the **TPL_APPLICATION** level.

Table 7-2 TPL Usage

<table>
<thead>
<tr>
<th>Task Priority Level</th>
<th>Usage</th>
</tr>
</thead>
</table>
Executing code can temporarily raise its priority level by calling the `EFI_BOOT_SERVICES.RaiseTPL()` function. Doing this masks event notifications from code running at equal or lower priority levels until the `EFI_BOOT_SERVICES.RestoreTPL()` function is called to reduce the priority to a level below that of the pending event notifications. There are restrictions on the TPL levels at which many UEFI service functions and protocol interface functions can execute. Table 7-3 summarizes the restrictions.

**Table 7-3 TPL Restrictions**

<table>
<thead>
<tr>
<th>Name</th>
<th>Restrictions</th>
<th>Task Priority Level</th>
</tr>
</thead>
<tbody>
<tr>
<td>ACPI Table Protocol</td>
<td>&lt;</td>
<td>TPL_NOTIFY</td>
</tr>
<tr>
<td>ARP</td>
<td>&lt;=</td>
<td>TPL_CALLBACK</td>
</tr>
<tr>
<td>ARP Service Binding</td>
<td>&lt;=</td>
<td>TPL_CALLBACK</td>
</tr>
<tr>
<td>Authentication Info</td>
<td>&lt;=</td>
<td>TPL_NOTIFY</td>
</tr>
<tr>
<td>Block I/O Protocol</td>
<td>&lt;=</td>
<td>TPL_CALLBACK</td>
</tr>
<tr>
<td>Block I/O 2 Protocol</td>
<td>&lt;=</td>
<td>TPL_CALLBACK</td>
</tr>
<tr>
<td>Bluetooth Host Controller</td>
<td>&lt;=</td>
<td>TPL_CALLBACK</td>
</tr>
<tr>
<td>Bluetooth IO Service Binding</td>
<td>&lt;=</td>
<td>TPL_CALLBACK</td>
</tr>
<tr>
<td>Bluetooth IO</td>
<td>&lt;=</td>
<td>TPL_CALLBACK</td>
</tr>
<tr>
<td>Bluetooth Attribute</td>
<td>&lt;=</td>
<td>TPL_CALLBACK</td>
</tr>
<tr>
<td>Bluetooth Configuration</td>
<td>&lt;=</td>
<td>TPL_CALLBACK</td>
</tr>
<tr>
<td>BluetoothLE Configuration</td>
<td>&lt;=</td>
<td>TPL_CALLBACK</td>
</tr>
<tr>
<td>Name</td>
<td>Restrictions</td>
<td>Task Priority Level</td>
</tr>
<tr>
<td>--------------------------------------------------</td>
<td>--------------</td>
<td>---------------------</td>
</tr>
<tr>
<td>CheckEvent()</td>
<td>&lt;</td>
<td>TPL_HIGH_LEVEL</td>
</tr>
<tr>
<td>CloseEvent()</td>
<td>&lt;</td>
<td>TPL_HIGH_LEVEL</td>
</tr>
<tr>
<td>CreateEvent()</td>
<td>&lt;</td>
<td>TPL_HIGH_LEVEL</td>
</tr>
<tr>
<td>Deferred Image Load Protocol</td>
<td>&lt;=</td>
<td>TPL_NOTIFY</td>
</tr>
<tr>
<td>Device Path Utilities</td>
<td>&lt;=</td>
<td>TPL_NOTIFY</td>
</tr>
<tr>
<td>Device Path From Text</td>
<td>&lt;=</td>
<td>TPL_NOTIFY</td>
</tr>
<tr>
<td>DHCP4 Service Binding</td>
<td>&lt;=</td>
<td>TPL_CALLBACK</td>
</tr>
<tr>
<td>DHCP4</td>
<td>&lt;=</td>
<td>TPL_CALLBACK</td>
</tr>
<tr>
<td>DHCP6</td>
<td>&lt;=</td>
<td>TPL_CALLBACK</td>
</tr>
<tr>
<td>DHCP6 Service Binding</td>
<td>&lt;=</td>
<td>TPL_CALLBACK</td>
</tr>
<tr>
<td>Disk I/O Protocol</td>
<td>&lt;=</td>
<td>TPL_CALLBACK</td>
</tr>
<tr>
<td>Disk I/O 2 Protocol</td>
<td>&lt;=</td>
<td>TPL_CALLBACK</td>
</tr>
<tr>
<td>DNS4 Service Binding</td>
<td>&lt;=</td>
<td>TPL_CALLBACK</td>
</tr>
<tr>
<td>DNS4</td>
<td>&lt;=</td>
<td>TPL_CALLBACK</td>
</tr>
<tr>
<td>DNS6 Service Binding</td>
<td>&lt;=</td>
<td>TPL_CALLBACK</td>
</tr>
<tr>
<td>DNS6</td>
<td>&lt;=</td>
<td>TPL_CALLBACK</td>
</tr>
<tr>
<td>Driver Health</td>
<td>&lt;=</td>
<td>TPL_NOTIFY</td>
</tr>
<tr>
<td>EAP</td>
<td>&lt;=</td>
<td>TPL_CALLBACK</td>
</tr>
<tr>
<td>EAP Configuration</td>
<td>&lt;=</td>
<td>TPL_CALLBACK</td>
</tr>
<tr>
<td>EAP Management</td>
<td>&lt;=</td>
<td>TPL_CALLBACK</td>
</tr>
<tr>
<td>EAP Management2</td>
<td>&lt;=</td>
<td>TPL_CALLBACK</td>
</tr>
<tr>
<td>EDID Active</td>
<td>&lt;=</td>
<td>TPL_NOTIFY</td>
</tr>
<tr>
<td>EDID Discovered</td>
<td>&lt;=</td>
<td>TPL_NOTIFY</td>
</tr>
<tr>
<td>EFI_SIMPLE_TEXT_INPUT_PROTOCOL</td>
<td>&lt;=</td>
<td>TPL_CALLBACK</td>
</tr>
<tr>
<td>EFI_SIMPLE_TEXT_INPUT_PROTOCOL.ReadKeyStroke</td>
<td>&lt;=</td>
<td>TPL_APPLICATION</td>
</tr>
<tr>
<td>EFI_SIMPLE_TEXT_INPUT_PROTOCOL.ReadKeyStroke</td>
<td>&lt;=</td>
<td>TPL_APPLICATION</td>
</tr>
<tr>
<td>EFI_SIMPLE_TEXT_INPUT_PROTOCOL.Reset</td>
<td>&lt;=</td>
<td>TPL_APPLICATION</td>
</tr>
<tr>
<td>EFI_SIMPLE_TEXT_INPUT_EX_PROTOCOL</td>
<td>&lt;=</td>
<td>TPL_CALLBACK</td>
</tr>
<tr>
<td>EFI_SIMPLE_TEXT_INPUT_EX_PROTOCOL.ReadKeyStrokeEx</td>
<td>&lt;=</td>
<td>TPL_APPLICATION</td>
</tr>
<tr>
<td>EFI_SIMPLE_TEXT_INPUT_EX_PROTOCOL.Reset</td>
<td>&lt;=</td>
<td>TPL_APPLICATION</td>
</tr>
<tr>
<td>Event Notification Levels</td>
<td>&gt;</td>
<td>TPL_APPLICATION</td>
</tr>
<tr>
<td></td>
<td>&lt;=</td>
<td>TPL_HIGH_LEVEL</td>
</tr>
<tr>
<td>Exit()</td>
<td>&lt;=</td>
<td>TPL_CALLBACK</td>
</tr>
<tr>
<td>ExitBootServices()</td>
<td>=</td>
<td>TPL_APPLICATION</td>
</tr>
<tr>
<td>Form Browser2 Protocol/SendForm</td>
<td>=</td>
<td>TPL.APPLICATION</td>
</tr>
<tr>
<td>Name</td>
<td>Restrictions</td>
<td>Task Priority Level</td>
</tr>
<tr>
<td>-------------------------------</td>
<td>--------------</td>
<td>---------------------</td>
</tr>
<tr>
<td>FTP</td>
<td>&lt;=</td>
<td>TPL_CALLBACK</td>
</tr>
<tr>
<td>Graphics Output EDID Override</td>
<td>&lt;=</td>
<td>TPL_NOTIFY</td>
</tr>
<tr>
<td>HII Protocols</td>
<td>&lt;=</td>
<td>TPL_NOTIFY</td>
</tr>
<tr>
<td>HTTP Service Binding</td>
<td>&lt;=</td>
<td>TPL_CALLBACK</td>
</tr>
<tr>
<td>HTTP</td>
<td>&lt;=</td>
<td>TPL_CALLBACK</td>
</tr>
<tr>
<td>HTTP Utilities</td>
<td>&lt;=</td>
<td>TPL_CALLBACK</td>
</tr>
<tr>
<td>IP4 Service Binding</td>
<td>&lt;=</td>
<td>TPL_CALLBACK</td>
</tr>
<tr>
<td>IP4</td>
<td>&lt;=</td>
<td>TPL_CALLBACK</td>
</tr>
<tr>
<td>IP4 Config</td>
<td>&lt;=</td>
<td>TPL_CALLBACK</td>
</tr>
<tr>
<td>IP4 Config2</td>
<td>&lt;=</td>
<td>TPL_CALLBACK</td>
</tr>
<tr>
<td>IP6</td>
<td>&lt;=</td>
<td>TPL_CALLBACK</td>
</tr>
<tr>
<td>IP6 Config</td>
<td>&lt;=</td>
<td>TPL_CALLBACK</td>
</tr>
<tr>
<td>IPSec Configuration</td>
<td>&lt;=</td>
<td>TPL_CALLBACK</td>
</tr>
<tr>
<td>iSCSI Initiator Name</td>
<td>&lt;=</td>
<td>TPL_NOTIFY</td>
</tr>
<tr>
<td>LoadImage()</td>
<td>&lt;</td>
<td>TPL_CALLBACK</td>
</tr>
<tr>
<td>Managed Network Service Binding</td>
<td>&lt;=</td>
<td>TPL_CALLBACK</td>
</tr>
<tr>
<td>Memory Allocation Services</td>
<td>&lt;=</td>
<td>TPL_NOTIFY</td>
</tr>
<tr>
<td>MTFTP4 Service Binding</td>
<td>&lt;=</td>
<td>TPL_CALLBACK</td>
</tr>
<tr>
<td>MTFTP4</td>
<td>&lt;=</td>
<td>TPL_CALLBACK</td>
</tr>
<tr>
<td>MTFTP6</td>
<td>&lt;=</td>
<td>TPL_CALLBACK</td>
</tr>
<tr>
<td>MTFTP6 Service Binding</td>
<td>&lt;=</td>
<td>TPL_CALLBACK</td>
</tr>
<tr>
<td>PXE Base Code Protocol</td>
<td>&lt;=</td>
<td>TPL_CALLBACK</td>
</tr>
<tr>
<td>Protocol Handler Services</td>
<td>&lt;=</td>
<td>TPL_NOTIFY</td>
</tr>
<tr>
<td>REST</td>
<td>&lt;=</td>
<td>TPL_CALLBACK</td>
</tr>
<tr>
<td>Serial I/O Protocol</td>
<td>&lt;=</td>
<td>TPL_CALLBACK</td>
</tr>
<tr>
<td>SetTimer()</td>
<td>&lt;</td>
<td>TPL_HIGH_LEVEL</td>
</tr>
<tr>
<td>SignalEvent()</td>
<td>&lt;=</td>
<td>TPL_HIGH_LEVEL</td>
</tr>
<tr>
<td>Simple File System Protocol</td>
<td>&lt;=</td>
<td>TPL_CALLBACK</td>
</tr>
<tr>
<td>Simple Network Protocol</td>
<td>&lt;=</td>
<td>TPL_CALLBACK</td>
</tr>
<tr>
<td>Simple Text Output Protocol</td>
<td>&lt;=</td>
<td>TPL_NOTIFY</td>
</tr>
<tr>
<td>Stall()</td>
<td>&lt;=</td>
<td>TPL_HIGH_LEVEL</td>
</tr>
<tr>
<td>StartImage()</td>
<td>&lt;</td>
<td>TPL_CALLBACK</td>
</tr>
<tr>
<td>Supplicant</td>
<td>&lt;=</td>
<td>TPL_CALLBACK</td>
</tr>
<tr>
<td>Tape IO</td>
<td>&lt;=</td>
<td>TPL_NOTIFY</td>
</tr>
<tr>
<td>TCP4 Service Binding</td>
<td>&lt;=</td>
<td>TPL_CALLBACK</td>
</tr>
<tr>
<td>TCP4</td>
<td>&lt;=</td>
<td>TPL_CALLBACK</td>
</tr>
<tr>
<td>TCP6</td>
<td>&lt;=</td>
<td>TPL_CALLBACK</td>
</tr>
</tbody>
</table>
EFI_BOOT_SERVICES.CreateEvent()

**Summary**
Creates an event.

**Prototype**
```c
typedef EFI_STATUS
    (EFIAPI *EFI_CREATE_EVENT) (
        IN UINT32 Type,
        IN EFI_TPL NotifyTpl,
        IN EFI_EVENT_NOTIFY NotifyFunction, OPTIONAL
        IN VOID *NotifyContext, OPTIONAL
        OUT EFI_EVENT *Event
    );
```

**Parameters**
- **Type**
The type of event to create and its mode and attributes. The `#define` statements in “Related Definitions” can be used to specify an event’s mode and attributes.
**NotifyTpl**

The task priority level of event notifications, if needed. See `EFI_BOOT_SERVICES.RaiseTPL()`.

**NotifyFunction**

Pointer to the event’s notification function, if any. See “Related Definitions.”

**NotifyContext**

Pointer to the notification function’s context; corresponds to parameter `Context` in the notification function.

**Event**

Pointer to the newly created event if the call succeeds; undefined otherwise.

---

**Related Definitions**

```c
//******************************************************************************
// EFI_EVENT
//******************************************************************************
typedef VOID *EFI_EVENT

//******************************************************************************
// Event Types
//******************************************************************************
// These types can be “ORed” together as needed – for example,
// EVT_TIMER might be “Ored” with EVT_NOTIFY_WAIT or
// EVT_NOTIFY_SIGNAL.
#define EVT_TIMER 0x80000000
#define EVT_RUNTIME 0x40000000
#define EVT_NOTIFY_WAIT 0x00000100
#define EVT_NOTIFY_SIGNAL 0x00000200
#define EVT_SIGNAL_EXIT_BOOT_SERVICES 0x00000201
#define EVT_SIGNAL_VIRTUAL_ADDRESS_CHANGE 0x60000202
```

- **EVT_TIMER**
  - The event is a timer event and may be passed to `EFI_BOOT_SERVICES.SetTimer()`. Note that timers only function during boot services time.

- **EVT_RUNTIME**
  - The event is allocated from runtime memory. If an event is to be signaled after the call to `EFI_BOOT_SERVICES.ExitBootServices()`, the event’s data structure and notification function need to be allocated from runtime memory. For more information, see `SetVirtualAddressMap()`.

- **EVT_NOTIFY_WAIT**
  - If an event of this type is not already in the signaled state, then the event’s `NotificationFunction` will be queued at the event’s `NotifyTpl` whenever the event is being waited on via `EFI_BOOT_SERVICES.WaitForEvent()` or `EFI_BOOT_SERVICES.CheckEvent()`.
EVT_NOTIFY_SIGNAL

The event’s NotifyFunction is queued whenever the event is signaled.

EVT_SIGNAL_EXIT_BOOT_SERVICES

This event is of type EVT_NOTIFY_SIGNAL. It should not be combined with any other event types. This event type is functionally equivalent to the EFI_EVENT_GROUP_EXIT_BOOT_SERVICES event group. Refer to EFI_EVENT_GROUP_EXIT_BOOT_SERVICES event group description in the EFI_BOOT_SERVICES.CreateEventEx() section below for additional details.

EVT_SIGNAL_VIRTUAL_ADDRESS_CHANGE

The event is to be notified by the system when SetVirtualAddressMap() is performed. This event type is a composite of EVT_NOTIFY_SIGNAL, EVT_RUNTIME, and EVT_RUNTIME_CONTEXT and should not be combined with any other event types.

//*******************************************************
// EFI_EVENT_NOTIFY
//*******************************************************
typedef
VOID
(EFIAPI *EFI_EVENT_NOTIFY) (  
    IN EFI_EVENT Event,
    IN VOID      *Context
);

Event

Event whose notification function is being invoked.

Context

Pointer to the notification function’s context, which is implementation-dependent. Context corresponds to NotifyContext in EFI_BOOT_SERVICES.CreateEventEx().

Description

The CreateEvent() function creates a new event of type Type and returns it in the location referenced by Event. The event’s notification function, context, and task priority level are specified by NotifyFunction, NotifyContext, and NotifyTpl, respectively.

Events exist in one of two states, “waiting” or “signaled.” When an event is created, firmware puts it in the “waiting” state. When the event is signaled, firmware changes its state to “signaled” and, if EVT_NOTIFY_SIGNAL is specified, places a call to its notification function in a FIFO queue. There is a queue for each of the “basic” task priority levels defined in Section 7.1 (TPL_CALLBACK, and TPL_NOTIFY). The functions in these queues are invoked in FIFO order, starting with the highest priority level queue and proceeding to the lowest priority queue that is unmasked by the current TPL. If the current TPL is equal to or greater than the queued notification, it will wait until the TPL is lowered via EFI_BOOT_SERVICES.RestoreTPL().

In a general sense, there are two “types” of events, synchronous and asynchronous. Asynchronous events are closely related to timers and are used to support periodic or timed interruption of program
execution. This capability is typically used with device drivers. For example, a network device driver that needs to poll for the presence of new packets could create an event whose type includes EVT_TIMER and then call the EFI_BOOT_SERVICES.SetTimer() function. When the timer expires, the firmware signals the event.

Synchronous events have no particular relationship to timers. Instead, they are used to ensure that certain activities occur following a call to a specific interface function. One example of this is the cleanup that needs to be performed in response to a call to the EFI_BOOT_SERVICES.ExitBootServices() function. ExitBootServices() can clean up the firmware since it understands firmware internals, but it cannot clean up on behalf of drivers that have been loaded into the system. The drivers have to do that themselves by creating an event whose type is EVT_SIGNAL_EXIT_BOOT_SERVICES and whose notification function is a function within the driver itself. Then, when ExitBootServices() has finished its cleanup, it signals each event of type EVT_SIGNAL_EXIT_BOOT_SERVICES.

Another example of the use of synchronous events occurs when an event of type EVT_SIGNAL_VIRTUAL_ADDRESS_CHANGE is used in conjunction with the SetVirtualAddressMap().

The EVT_NOTIFY_WAIT and EVT_NOTIFY_SIGNAL flags are exclusive. If neither flag is specified, the caller does not require any notification concerning the event and the NotifyTpl, NotifyFunction, and NotifyContext parameters are ignored. If EVT_NOTIFY_WAIT is specified and the event is not in the signaled state, then the EVT_NOTIFY_WAIT notify function is queued whenever a consumer of the event is waiting for the event (via EFI_BOOT_SERVICES.WaitForEvent() or EFI_BOOT_SERVICES.CheckEvent()). If the EVT_NOTIFY_SIGNAL flag is specified then the event’s notify function is queued whenever the event is signaled.
**Note:** Because its internal structure is unknown to the caller, Event cannot be modified by the caller. The only way to manipulate it is to use the published event interfaces.

### Status Codes Returned

<table>
<thead>
<tr>
<th>Code</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>EFI_SUCCESS</td>
<td>The event structure was created.</td>
</tr>
<tr>
<td>EFI_INVALID_PARAMETER</td>
<td>One of the parameters has an invalid value.</td>
</tr>
<tr>
<td>EFI_INVALID_PARAMETER</td>
<td>Event is NULL.</td>
</tr>
<tr>
<td>EFI_INVALID_PARAMETER</td>
<td>Type has an unsupported bit set.</td>
</tr>
<tr>
<td>EFI_INVALID_PARAMETER</td>
<td>Type has both EVT_NOTIFY_SIGNAL and EVT_NOTIFY_WAIT set.</td>
</tr>
<tr>
<td>EFI_INVALID_PARAMETER</td>
<td>Type has either EVT_NOTIFY_SIGNAL or EVT_NOTIFY_WAIT set and NotifyFunction is NULL.</td>
</tr>
<tr>
<td>EFI_INVALID_PARAMETER</td>
<td>Type has either EVT_NOTIFY_SIGNAL or EVT_NOTIFY_WAIT set and NotifyTpl is not a supported TPL level.</td>
</tr>
<tr>
<td>EFI_OUT_OF_RESOURCES</td>
<td>The event could not be allocated.</td>
</tr>
</tbody>
</table>

### EFI_BOOT_SERVICES.CreateEventEx()

**Summary**

Creates an event in a group.

**Prototype**

```c
typedef EFI_STATUS
(EFIAPI *EFI_CREATE_EVENT_EX) (  
    IN UINT32 Type,  
    IN EFI_TPL NotifyTpl,  
    IN EFI_EVENT_NOTIFY NotifyFunction OPTIONAL,  
    IN CONST VOID *NotifyContext OPTIONAL,  
    IN CONST EFI_GUID *EventGroup OPTIONAL,  
    OUT EFI_EVENT *Event  
);
```

**Parameters**

- **Type**
  The type of event to create and its mode and attributes.
- **NotifyTpl**
  The task priority level of event notifications, if needed. See EFI_BOOT_SERVICES.RaiseTPL().
- **NotifyFunction**
  Pointer to the event’s notification function, if any.
- **NotifyContext**
  Pointer to the notification function’s context; corresponds to parameter Context in the notification function.
### EventGroup
Pointer to the unique identifier of the group to which this event belongs. If this is **NULL** then the function behaves as if the parameters were passed to **CreateEvent**.

### Event
Pointer to the newly created event if the call succeeds; undefined otherwise.

**Description**

The **CreateEventEx** function creates a new event of type **Type** and returns it in the specified location indicated by **Event**. The event’s notification function, context and task priority are specified by **NotifyFunction**, **NotifyContext**, and **NotifyTpl**, respectively. The event will be added to the group of events identified by **EventGroup**.

If no group is specified by **EventGroup**, then this function behaves as if the same parameters had been passed to **CreateEvent**.

Event groups are collections of events identified by a shared EFI_GUID where, when one member event is signaled, all other events are signaled and their individual notification actions are taken (as described in **CreateEvent**). All events are guaranteed to be signaled before the first notification action is taken. All notification functions will be executed in the order specified by their **NotifyTpl**.

A single event can only be part of a single event group. An event may be removed from an event group by using **CloseEvent**.

The **Type** of an event uses the same values as defined in **CreateEvent** except that **EVT_SIGNAL_EXIT_BOOT_SERVICES** and **EVT_SIGNAL_VIRTUAL_ADDRESS_CHANGE** are not valid.

If **Type** has **EVT_NOTIFY_SIGNAL** or **EVT_NOTIFY_WAIT**, then **NotifyFunction** must be non-**NULL** and **NotifyTpl** must be a valid task priority level. Otherwise these parameters are ignored.

More than one event of type **EVT_TIMER** may be part of a single event group. However, there is no mechanism for determining which of the timers was signaled.

**Configuration Table Groups**

The GUID for a configuration table also defines a corresponding event group GUID with the same value. If the data represented by a configuration table is changed, **InstallConfigurationTable()** should be called. When **InstallConfigurationTable()** is called, the corresponding event is signaled. When this event is signaled, any components that cache information from the configuration table can optionally update their cached state.

For example, **EFI_ACPI_TABLE_GUID** defines a configuration table for ACPI data. When ACPI data is changed, **InstallConfigurationTable()** is called. During the execution of **InstallConfigurationTable()**, a corresponding event group with **EFI_ACPI_TABLE_GUID** is signaled, allowing an application to invalidate any cached ACPI data.
Pre-Defined Event Groups
This section describes the pre-defined event groups used by the UEFI specification.

**EFI_EVENT_GROUP_EXIT_BOOT_SERVICES**

This event group is notified by the system when ExitBootServices() is invoked after notifying EFI_EVENT_GROUP_BEFORE_EXIT_BOOT_SERVICES event group. This event group is functionally equivalent to the EVT_SIGNAL_EXIT_BOOT_SERVICES flag for the Type argument of CreateEvent. The notification function for this event must comply with the following requirements:

- The notification function is not allowed to use the Memory Allocation Services, or call any functions that use the Memory Allocation Services, because these services modify the current memory map.

*NOTE:* Since consumer of the service does not necessarily knows if the service uses memory allocation services, this requirement is effectively a mandate to reduce usage of any external services (services implemented outside of the driver owning the notification function) to an absolute minimum required to perform an orderly transition to a runtime environment. Usage of the external services may yield unexpected results. Since UEFI specification does not guarantee any given order of notification function invocation, a notification function consuming the service may be invoked before or after the notification function of the driver providing the service. As a result, a service being called by the notification function may exhibit boot time behavior or a runtime behavior (which is undefined for a pure boot services).

- The notification function must not depend on timer events since timer services will be deactivated before any notification functions are called.

Refer to **EFI_BOOT_SERVICES.ExitBootServices()** below for additional details.

**EFI_EVENT_GROUP_BEFORE_EXIT_BOOT_SERVICES**

This event group is notified by the system when ExitBootServices() is invoked right before notifying EFI_EVENT_GROUP_EXIT_BOOT_SERVICES event group. The event presents the last opportunity to use firmware interfaces in the boot environment.

The notification function for this event must not depend on any kind of delayed processing (processing that happens in a timer callback beyond the time span of the notification function) because system firmware deactivates timer services right after dispatching handlers for this event group.

Refer to **EFI_BOOT_SERVICES.ExitBootServices()** below for additional details.

**EFI_EVENT_GROUP_VIRTUAL_ADDRESS_CHANGE**

This event group is notified by the system when SetVirtualAddressMap() is invoked. This is functionally equivalent to the EVT_SIGNAL_VIRTUAL_ADDRESS_CHANGE flag for the Type argument of CreateEvent.

**EFI_EVENT_GROUP_MEMORY_MAP_CHANGE**

This event group is notified by the system when the memory map has changed. The notification function for this event should not use Memory Allocation Services to avoid reentrancy complications.
**EFI_EVENT_GROUP_READY_TO_BOOT**
This event group is notified by the system right before notifying
**EFI_EVENT_GROUP_AFTER_READY_TO_BOOT** event group when the Boot Manager
is about to load and execute a boot option. The event group presents the last chance
to modify device or system configuration prior to passing control to a boot option.

**EFI_EVENT_GROUP_AFTER_READY_TO_BOOT**
This event group is notified by the system immediately after notifying
**EFI_EVENT_GROUP_READY_TO_BOOT** event group when the Boot Manager is about to load and
execute a boot option. The event group presents the last chance to survey device or system configuration
prior to passing control to a boot option.

**EFI_EVENT_GROUP_RESET_SYSTEM**
This event group is notified by the system when ResetSystem() is invoked and the
system is about to be reset. The event group is only notified prior to
ExitBootServices() invocation.

Related Definitions
**EFI_EVENT** is defined in CreateEvent.
**EVT_SIGNAL_EXIT_BOOT_SERVICES** and **EVT_SIGNAL_VIRTUAL_ADDRESS_CHANGE** are defined in
CreateEvent.

```c
#define EFI_EVENT_GROUP_EXIT_BOOT_SERVICES
   {0x27abf055, 0xb1b8, 0x4c26, 0x80, 0x48, 0x74, 0x8f, 0x37,\
    0xba, 0xa2, 0xdf}\n
#define EFI_EVENT_GROUP_BEFORE_EXIT_BOOT_SERVICES
   { 0x8be0e274, 0x3970, 0x4b44, { 0x80, 0xc5, 0x1a, 0xb9, 0x50, 0x2f, 0x3b,\n    0xfc } }\n
#define EFI_EVENT_GROUP_VIRTUAL_ADDRESS_CHANGE
   {0x13fa7698, 0xc831, 0x49c7, 0x87, 0xea, 0x8f, 0x43, 0xfc,\n    0xc2, 0x51, 0x96}\n
#define EFI_EVENT_GROUP_MEMORY_MAP_CHANGE
   {0x78bee926, 0x692f, 0x48fd, 0x9e, 0xdb, 0x1, 0x42, 0x2e,\n    0xf0, 0xd7, 0xab}\n
#define EFI_EVENT_GROUP_READY_TO_BOOT
   {0xce88fb3, 0x4bd7, 0x4679, 0x87, 0xa8, 0xa8, 0xd8, 0xde,\n    0xe5,0xd, 0x2b}\n
#define EFI_EVENT_GROUP_AFTER_READY_TO_BOOT
   { 0x3a2a00ad, 0x98b9, 0x4cdf, { 0xa4, 0x78, 0x70, 0x27, 0x77, 0xf1, 0xc1,\n    0xb } }\n```
#define EFI_EVENT_GROUP_RESET_SYSTEM \
{ 0x62da6a56, 0x13fb, 0x485a, { 0xa8, 0xda, 0xa3, 0xdd, 0x79, 0x12, 0xcb, 0x6b } }

Status Codes Returned

<table>
<thead>
<tr>
<th>Code</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>EFI_SUCCESS</td>
<td>The event structure was created.</td>
</tr>
<tr>
<td>EFI_INVALID_PARAMETER</td>
<td>One of the parameters has an invalid value.</td>
</tr>
<tr>
<td>EFI_INVALID_PARAMETER</td>
<td>Event is NULL</td>
</tr>
<tr>
<td>EFI_INVALID_PARAMETER</td>
<td>Type has an unsupported bit set.</td>
</tr>
<tr>
<td>EFI_INVALID_PARAMETER</td>
<td>Type has both EVT_NOTIFY_SIGNAL and EVT_NOTIFY_WAIT set.</td>
</tr>
<tr>
<td>EFI_INVALID_PARAMETER</td>
<td>Type has either EVT_NOTIFY_SIGNAL or EVT_NOTIFY_WAIT set and NotifyFunction is NULL.</td>
</tr>
<tr>
<td>EFI_INVALID_PARAMETER</td>
<td>Type has either EVT_NOTIFY_SIGNAL or EVT_NOTIFY_WAIT set and NotifyTpl is not a supported TPL level.</td>
</tr>
<tr>
<td>EFI_OUT_OF_RESOURCES</td>
<td>The event could not be allocated.</td>
</tr>
</tbody>
</table>

EFI_BOOT_SERVICES.CloseEvent()

Summary
Closes an event.

Prototype

typedef
EFI_STATUS
(EIFIAPI *EFI_CLOSE_EVENT) ( 
  IN EFI_EVENT Event 
);

Parameters
Event
The event to close. Type EFI_EVENT is defined in the CreateEvent() function description.

Description
The CloseEvent() function removes the caller’s reference to the event, removes it from any event group to which it belongs, and closes it. Once the event is closed, the event is no longer valid and may not be used on any subsequent function calls. If Event was registered with RegisterProtocolNotify() then CloseEvent() will remove the corresponding registration. It is safe to call CloseEvent() within the corresponding notify function.
Status Codes Returned

<table>
<thead>
<tr>
<th>Code</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>EFI_SUCCESS</td>
<td>The event has been closed.</td>
</tr>
</tbody>
</table>

**EFI_BOOT_SERVICES.SignalEvent()**

**Summary**
Signals an event.

**Prototype**
```
typedef EFI_STATUS
    (EFIAPI *EFI_SIGNAL_EVENT) (
    IN EFI_EVENT Event
    );
```

**Parameters**
- **Event**
  The event to signal. Type EFI_EVENT is defined in the EFI_BOOT_SERVICES.CheckEvent() function description.

**Description**
The supplied Event is placed in the signaled state. If Event is already in the signaled state, then EFI_SUCCESS is returned. If Event is of type EVT_NOTIFY_SIGNAL, then the event’s notification function is scheduled to be invoked at the event’s notification task priority level. SignalEvent() may be invoked from any task priority level.

If the supplied Event is a part of an event group, then all of the events in the event group are also signaled and their notification functions are scheduled.

When signaling an event group, it is possible to create an event in the group, signal it and then close the event to remove it from the group. For example:
```c
 EFI_EVENT Event;
 EFI_GUID gMyEventGroupGuid = EFI_MY_EVENT_GROUP_GUID;
gBS->CreateEventEx ( 
  0,
  0,
  NULL,
  NULL,
  &gMyEventGroupGuid,
  &Event
);

gBS->SignalEvent (Event);
 gBS->CloseEvent (Event);
```

### Status Codes Returned

<table>
<thead>
<tr>
<th>Status Code</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>EFI_SUCCESS</td>
<td>The event was signaled.</td>
</tr>
</tbody>
</table>

### EFI_BOOT_SERVICES.WaitForEvent()

**Summary**

Stops execution until an event is signaled.

**Prototype**

```c
typedef
 EFI_STATUS
 (EFIAPI *EFI_WAIT_FOR_EVENT) ( 
  IN UINTN NumberOfEvents,
  IN EFI_EVENT *Event,
  OUT UINTN *Index
);
```

**Parameters**

- **NumberOfEvents**
  - The number of events in the Event array.
- **Event**
  - An array of EFI_EVENT. Type EFI_EVENT is defined in the CreateEvent() function description.
- **Index**
  - Pointer to the index of the event which satisfied the wait condition.

**Description**

This function must be called at priority level TPL_APPLICATION. If an attempt is made to call it at any other priority level, EFI_UNSUPPORTED is returned.

The list of events in the Event array are evaluated in order from first to last, and this evaluation is repeated until an event is signaled or an error is detected. The following checks are performed on each event in the Event array.

- If an event is of type EVT_NOTIFY_SIGNAL, then EFI_INVALID_PARAMETER is returned and Index indicates the event that caused the failure.
• If an event is in the signaled state, the signaled state is cleared and **EFI_SUCCESS** is returned, and **Index** indicates the event that was signaled.

• If an event is not in the signaled state but does have a notification function, the notification function is queued at the event’s notification task priority level. If the execution of the event’s notification function causes the event to be signaled, then the signaled state is cleared, **EFI_SUCCESS** is returned, and **Index** indicates the event that was signaled.

To wait for a specified time, a timer event must be included in the **Event** array.

To check if an event is signaled without waiting, an already signaled event can be used as the last event in the list being checked, or the **CheckEvent()** interface may be used.

### Status Codes Returned

<table>
<thead>
<tr>
<th>Code</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>EFI_SUCCESS</strong></td>
<td>The event indicated by <strong>Index</strong> was signaled.</td>
</tr>
<tr>
<td><strong>EFI_INVALID_PARAMETER</strong></td>
<td>NumberOfEvents is 0.</td>
</tr>
<tr>
<td><strong>EFI_INVALID_PARAMETER</strong></td>
<td>The event indicated by <strong>Index</strong> is of type <strong>EVT_NOTIFY_SIGNAL</strong>.</td>
</tr>
<tr>
<td><strong>EFI_UNSUPPORTED</strong></td>
<td>The current TPL is not <strong>TPL_APPLICATION</strong>.</td>
</tr>
</tbody>
</table>

### EFI_BOOT_SERVICES.CheckEvent()

**Summary**

Checks whether an event is in the signaled state.

**Prototype**

```c
typedef
EFI_STATUS
(EIFIAPI *EFI_CHECK_EVENT) (
    IN EFI_EVENT Event
);
```

**Parameters**

- **Event** The event to check. Type **EFI_EVENT** is defined in the **CreateEvent()** function description.

**Description**

The **CheckEvent()** function checks to see whether **Event** is in the signaled state. If **Event** is of type **EVT_NOTIFY_SIGNAL**, then **EFI_INVALID_PARAMETER** is returned. Otherwise, there are three possibilities:

- If **Event** is in the signaled state, it is cleared and **EFI_SUCCESS** is returned.
- If **Event** is not in the signaled state and has no notification function, **EFI_NOT_READY** is returned.
- If **Event** is not in the signaled state but does have a notification function, the notification function is queued at the event’s notification task priority level. If the execution of the notification function causes **Event** to be signaled, then the signaled state is cleared and **EFI_SUCCESS** is returned; if the **Event** is not signaled, then **EFI_NOT_READY** is returned.
Status Codes Returned

<table>
<thead>
<tr>
<th>Status Code</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>EFI_SUCCESS</td>
<td>The event is in the signaled state.</td>
</tr>
<tr>
<td>EFI_NOT_READY</td>
<td>The event is not in the signaled state.</td>
</tr>
<tr>
<td>EFI_INVALID_PARAMETER</td>
<td>Event is of type EVT_NOTIFY_SIGNAL.</td>
</tr>
</tbody>
</table>

**EFI_BOOT_SERVICES.SetTimer()**

**Summary**
Sets the type of timer and the trigger time for a timer event.

**Prototype**

typedef
EFI_STATUS
(EFIAPI "EFI_SET_TIMER") (        
IN EFI_EVENT Event,        
IN EFI_TIMER_DELAY Type,        
IN UINT64 TriggerTime        
);

**Parameters**

- **Event**
The timer event that is to be signaled at the specified time. Type EFI_EVENT is defined in the CreateEvent() function description.

- **Type**
The type of time that is specified in TriggerTime. See the timer delay types in “Related Definitions.”

- **TriggerTime**
The number of 100ns units until the timer expires. A TriggerTime of 0 is legal. If Type is TimerRelative and TriggerTime is 0, then the timer event will be signaled on the next timer tick. If Type is TimerPeriodic and TriggerTime is 0, then the timer event will be signaled on every timer tick.

**Related Definitions**

```c
//**********************************************************************
//EFI_TIMER_DELAY
//**********************************************************************
typedef enum {
    TimerCancel, 
    TimerPeriodic,  
    TimerRelative
} EFI_TIMER_DELAY;
```

- **TimerCancel**
The event's timer setting is to be cancelled and no timer trigger is to be set. TriggerTime is ignored when canceling a timer.

- **TimerPeriodic**
The event is to be signaled periodically at TriggerTime intervals from the current time. This is the only timer trigger Type for which
the event timer does not need to be reset for each notification. All other timer trigger types are “one shot.”

**TimerRelative**  
The event is to be signaled in **TriggerTime** 100ns units.

**Description**  
The **SetTimer()** function cancels any previous time trigger setting for the event, and sets the new trigger time for the event. This function can only be used on events of type **EVT_TIMER**.

**Status Codes Returned**

<table>
<thead>
<tr>
<th>Status Code</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>EFI_SUCCESS</td>
<td>The event has been set to be signaled at the requested time.</td>
</tr>
<tr>
<td>EFI_INVALID_PARAMETER</td>
<td><strong>Event</strong> or <strong>Type</strong> is not valid.</td>
</tr>
</tbody>
</table>

**EFI_BOOT_SERVICES.RaiseTPL()**

**Summary**  
Raises a task’s priority level and returns its previous level.

**Prototype**

```c
typedef EFI_TPL
(EFIAPIC _EFI_RAISE_TPL) (      
    IN EFI_TPL NewTpl
);
```

**Parameters**

- **NewTpl**  
The new task priority level. It must be greater than or equal to the current task priority level. See “Related Definitions.”

**Related Definitions**

```c
//*******************************************************
// EFI_TPL
//*******************************************************
typedef UINTN   EFI_TPL

/******************************************************************************
// Task Priority Levels
//******************************************************************************
#define TPL_APPLICATION  4
#define TPL_CALLBACK     8
#define TPL_NOTIFY      16
#define TPL_HIGH_LEVEL   31
```

**Description**  
The **EFI_BOOT_SERVICES.RaiseTPL()** function raises the priority of the currently executing task and returns its previous priority level.
Only three task priority levels are exposed outside of the firmware during boot services execution. The first is `TPL_APPLICATION` where all normal execution occurs. That level may be interrupted to perform various asynchronous interrupt style notifications, which occur at the `TPL_CALLBACK` or `TPL_NOTIFY` level. By raising the task priority level to `TPL_NOTIFY` such notifications are masked until the task priority level is restored, thereby synchronizing execution with such notifications. Synchronous blocking I/O functions execute at `TPL_NOTIFY`. `TPL_CALLBACK` is the typically used for application level notification functions. Device drivers will typically use `TPL_CALLBACK` or `TPL_NOTIFY` for their notification functions. Applications and drivers may also use `TPL_NOTIFY` to protect data structures in critical sections of code.

The caller must restore the task priority level with `EFI_BOOT_SERVICES.RestoreTPL()` to the previous level before returning.

**Note:** If `NewTpl` is below the current TPL level, then the system behavior is indeterminate. Additionally, only `TPL_APPLICATION`, `TPL_CALLBACK`, `TPL_NOTIFY`, and `TPL_HIGH_LEVEL` may be used. All other values are reserved for use by the firmware; using them will result in unpredictable behavior. Good coding practice dictates that all code should execute at its lowest possible TPL level, and the use of TPL levels above `TPL_APPLICATION` must be minimized. Executing at TPL levels above `TPL_APPLICATION` for extended periods of time may also result in unpredictable behavior.

**Status Codes Returned**

Unlike other UEFI interface functions, `EFI_BOOT_SERVICES.RaiseTPL()` does not return a status code. Instead, it returns the previous task priority level, which is to be restored later with a matching call to `RestoreTPL()`.

`EFI_BOOT_SERVICES.RestoreTPL()`

**Summary**

Restores a task’s priority level to its previous value.

**Prototype**

```c
typedef VOID (EFIAPI *EFI_RESTORE_TPL) (IN EFI_TPL OldTpl)
```

**Parameters**

- `OldTpl` The previous task priority level to restore (the value from a previous, matching call to `EFI_BOOT_SERVICES.RaiseTPL()`). Type `EFI_TPL` is defined in the `RaiseTPL()` function description.

**Description**

The `RestoreTPL()` function restores a task’s priority level to its previous value. Calls to `RestoreTPL()` are matched with calls to `RaiseTPL()`.
Note: If OldTpl is above the current TPL level, then the system behavior is indeterminate. Additionally, only TPL_APPLICATION, TPL_CALLBACK, TPL_NOTIFY, and TPL_HIGH_LEVEL may be used. All other values are reserved for use by the firmware; using them will result in unpredictable behavior. Good coding practice dictates that all code should execute at its lowest possible TPL level, and the use of TPL levels above TPL_APPLICATION must be minimized. Executing at TPL levels above TPL_APPLICATION for extended periods of time may also result in unpredictable behavior.

Status Codes Returned
None.

7.2 Memory Allocation Services

The functions that make up Memory Allocation Services are used during preboot to allocate and free memory, and to obtain the system’s memory map. See Table 7-4.

Table 7-4 Memory Allocation Functions

<table>
<thead>
<tr>
<th>Name</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>AllocatePages</td>
<td>Boot</td>
<td>Allocates pages of a particular type.</td>
</tr>
<tr>
<td>FreePages</td>
<td>Boot</td>
<td>Frees allocated pages.</td>
</tr>
<tr>
<td>GetMemoryMap</td>
<td>Boot</td>
<td>Returns the current boot services memory map and memory map key.</td>
</tr>
<tr>
<td>AllocatePool</td>
<td>Boot</td>
<td>Allocates a pool of a particular type.</td>
</tr>
<tr>
<td>FreePool</td>
<td>Boot</td>
<td>Frees allocated pool.</td>
</tr>
</tbody>
</table>

The way in which these functions are used is directly related to an important feature of UEFI memory design. This feature, which stipulates that EFI firmware owns the system’s memory map during preboot, has three major consequences:

- During preboot, all components (including executing EFI images) must cooperate with the firmware by allocating and freeing memory from the system with the functions EFI_BOOT_SERVICES.AllocatePages(), EFI_BOOT_SERVICES.AllocatePool(), EFI_BOOT_SERVICES.FreePages(), and EFI_BOOT_SERVICES.FreePool(). The firmware dynamically maintains the memory map as these functions are called.
- During preboot, an executing EFI Image must only use the memory it has allocated.
- Before an executing EFI image exits and returns control to the firmware, it must free all resources it has explicitly allocated. This includes all memory pages, pool allocations, open file handles, etc. Memory allocated by the firmware to load an image is freed by the firmware when the image is unloaded.

This specification describes numerous memory buffers that are allocated by a service, where it is the caller’s responsibility to free the allocated memory. Unless stated otherwise in this specification, it is assumed that such memory buffers are allocated with AllocatePool() and freed with FreePool().

When memory is allocated, it is “typed” according to the values in EFI_MEMORY_TYPE (see the description for EFI_BOOT_SERVICES.AllocatePages()). Some of the types have a different usage before EFI_BOOT_SERVICES.ExitBootServices() is called than they do afterwards. Table 7-5 lists each type and its usage before the call; Table 7-6 lists each type and its usage after the call. The system
firmware must follow the processor-specific rules outlined in Section 2.3.2 and Section 2.3.4 in the layout of the EFI memory map to enable the OS to make the required virtual mappings.

Table 7-5 Memory Type Usage before ExitBootServices()

<table>
<thead>
<tr>
<th>Mnemonic</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>EfiReservedMemoryType</td>
<td>Not usable.</td>
</tr>
<tr>
<td>EfiLoaderCode</td>
<td>The code portions of a loaded UEFI application.</td>
</tr>
<tr>
<td>EfiLoaderData</td>
<td>The data portions of a loaded UEFI application and the default data allocation type used by a UEFI application to allocate pool memory.</td>
</tr>
<tr>
<td>EfiBootServicesCode</td>
<td>The code portions of a loaded UEFI Boot Service Driver.</td>
</tr>
<tr>
<td>EfiBootServicesData</td>
<td>The data portions of a loaded UEFI Boot Service Driver, and the default data allocation type used by a UEFI Boot Service Driver to allocate pool memory.</td>
</tr>
<tr>
<td>EfiRuntimeServicesCode</td>
<td>The code portions of a loaded UEFI Runtime Driver.</td>
</tr>
<tr>
<td>EfiRuntimeServicesData</td>
<td>The data portions of a loaded UEFI Runtime Driver and the default data allocation type used by a UEFI Runtime Driver to allocate pool memory.</td>
</tr>
<tr>
<td>EfiConventionalMemory</td>
<td>Free (unallocated) memory.</td>
</tr>
<tr>
<td>EfiUnusableMemory</td>
<td>Memory in which errors have been detected.</td>
</tr>
<tr>
<td>EfiACPIReclaimMemory</td>
<td>Memory that holds the ACPI tables.</td>
</tr>
<tr>
<td>EfiACPIMemoryNVS</td>
<td>Address space reserved for use by the firmware.</td>
</tr>
<tr>
<td>EfiMemoryMappedIO</td>
<td>Used by system firmware to request that a memory-mapped IO region be mapped by the OS to a virtual address so it can be accessed by EFI runtime services.</td>
</tr>
<tr>
<td>EfiMemoryMappedIOPortSpace</td>
<td>System memory-mapped IO region that is used to translate memory cycles to IO cycles by the processor.</td>
</tr>
<tr>
<td>EfiPalCode</td>
<td>Address space reserved by the firmware for code that is part of the processor.</td>
</tr>
<tr>
<td>EfiPersistentMemory</td>
<td>A memory region that operates as EfiConventionalMemory. However, it happens to also support byte-addressable non-volatility.</td>
</tr>
<tr>
<td>EfiUnacceptedMemoryType</td>
<td>A memory region that represents unaccepted memory, that must be accepted by the boot target before it can be used. Unless otherwise noted, all other EFI memory types are accepted. For platforms that support unaccepted memory, all unaccepted valid memory will be reported as unaccepted in the memory map. Unreported physical address ranges must be treated as not-present memory.</td>
</tr>
</tbody>
</table>

Note: There is only one region of type EfiMemoryMappedIOPortSpace defined in the architecture for Itanium-based platforms. As a result, there should be one and only one region of type EfiMemoryMappedIOPortSpace in the EFI memory map of an Itanium-based platform.

Table 7-6 Memory Type Usage after ExitBootServices()

<table>
<thead>
<tr>
<th>Mnemonic</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>EfiReservedMemoryType</td>
<td>Not usable.</td>
</tr>
<tr>
<td>EfiLoaderCode</td>
<td>The UEFI OS Loader and/or OS may use this memory as they see fit. Note: the UEFI OS loader that called EFI_BOOT_SERVICES.ExitBootServices() is utilizing one or more EfiLoaderCode ranges.</td>
</tr>
<tr>
<td>EfiPersistMemory</td>
<td>A memory region that operates as EfiConventionalMemory. However, it happens to also support byte-addressable non-volatility.</td>
</tr>
</tbody>
</table>


<table>
<thead>
<tr>
<th>Memory Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>EfiLoaderData</td>
<td>The Loader and/or OS may use this memory as they see fit. Note: the OS loader that called <strong>ExitBootServices()</strong> is utilizing one or more <strong>EfiLoaderData</strong> ranges.</td>
</tr>
<tr>
<td>EfiBootServicesCode</td>
<td>Memory available for general use.</td>
</tr>
<tr>
<td>EfiBootServicesData</td>
<td>Memory available for general use.</td>
</tr>
<tr>
<td>EfiRuntimeServicesCode</td>
<td>The memory in this range is to be preserved by the UEFI OS loader and OS in the working and ACPI S1–S3 states.</td>
</tr>
<tr>
<td>EfiRuntimeServicesData</td>
<td>The memory in this range is to be preserved by the UEFI OS loader and OS in the working and ACPI S1–S3 states.</td>
</tr>
<tr>
<td>EfiConventionalMemory</td>
<td>Memory available for general use.</td>
</tr>
<tr>
<td>EfiUnusableMemory</td>
<td>Memory that contains errors and is not to be used.</td>
</tr>
<tr>
<td>EfiACPIReclaimMemory</td>
<td>This memory is to be preserved by the UEFI OS loader and OS until ACPI is enabled. Once ACPI is enabled, the memory in this range is available for general use.</td>
</tr>
<tr>
<td>EfiACPIMemoryNVS</td>
<td>This memory is to be preserved by the UEFI OS loader and OS in the working and ACPI S1–S3 states.</td>
</tr>
<tr>
<td>EfiMemoryMappedIO</td>
<td>This memory is not used by the OS. All system memory-mapped IO information should come from ACPI tables.</td>
</tr>
<tr>
<td>EfiMemoryMappedIOPortSpace</td>
<td>This memory is not used by the OS. All system memory-mapped IO port space information should come from ACPI tables.</td>
</tr>
<tr>
<td>EfiPalCode</td>
<td>This memory is to be preserved by the UEFI OS loader and OS in the working and ACPI S1–S4 states. This memory may also have other attributes that are defined by the processor implementation.</td>
</tr>
<tr>
<td>EfiPersistentMemory</td>
<td>A memory region that operates as <strong>EfiConventionalMemory</strong>. However, it happens to also support byte-addressable non-volatility.</td>
</tr>
<tr>
<td>EfiUnacceptedMemoryType</td>
<td>A memory region that represents unaccepted memory, that must be accepted by the boot target before it can be used. Unless otherwise noted, all other EFI memory types are accepted. For platforms that support unaccepted memory, all unaccepted valid memory will be reported as unaccepted in the memory map. Unreported physical address ranges must be treated as not-present memory.</td>
</tr>
</tbody>
</table>

**Note:** An image that calls **ExitBootServices()** (i.e., a UEFI OS Loader) first calls EFI_BOOT_SERVICES.GetMemoryMap() to obtain the current memory map. Following the **ExitBootServices()** call, the image implicitly owns all unused memory in the map. This includes memory types EfiLoaderCode, EfiLoaderData, EfiBootServicesCode, EfiBootServicesData, and EfiConventionalMemory. A UEFI OS Loader and OS must preserve the memory marked as EfiRuntimeServicesCode and EfiRuntimeServicesData.

**EFI_BOOT_SERVICES.AllocatePages()**

**Summary**

Allocates memory pages from the system.
Prototype

```c
typedef EFI_STATUS (EFIAPI *EFI_ALLOCATE_PAGES) (IN EFI_ALLOCATE_TYPE Type,
                                               IN EFI_MEMORY_TYPE MemoryType,
                                               IN UINTN Pages,
                                               IN OUT EFI_PHYSICAL_ADDRESS *Memory);
```

Parameters

- **Type**: The type of allocation to perform. See “Related Definitions.”
- **MemoryType**: The type of memory to allocate. The type `EFI_MEMORY_TYPE` is defined in “Related Definitions” below. These memory types are also described in more detail in Table 7-5 and Table 7-6. Normal allocations (that is, allocations by any UEFI application) are of type `EfiLoaderData`. `MemoryType` values in the range 0x70000000..0x7FFFFFFF are reserved for OEM use. `MemoryType` values in the range 0x80000000..0xFFFFFFFF are reserved for use by UEFI OS loaders that are provided by operating system vendors.
- **Pages**: The number of contiguous 4 KiB pages to allocate.
- **Memory**: Pointer to a physical address. On input, the way in which the address is used depends on the value of `Type`. See “Description” for more information. On output the address is set to the base of the page range that was allocated. See “Related Definitions.”

Note: UEFI Applications, UEFI Drivers, and UEFI OS Loaders must not allocate memory of types `EfiReservedMemoryType`, `EfiMemoryMappedIO`, and `EfiUnacceptedMemoryType`.

Related Definitions

```c
//*******************************************************
//EFI_ALLOCATE_TYPE
//*******************************************************
// These types are discussed in the “Description” section below.
typedef enum {
   AllocateAnyPages,
   AllocateMaxAddress,
   AllocateAddress,
   MaxAllocateType
} EFI_ALLOCATE_TYPE;
```

```c
//*******************************************************
//EFI_MEMORY_TYPE
//*******************************************************
// These type values are discussed in Table 7-5 and Table 7-6.
typedef enum {
```
typedef UINT64 EFI_PHYSICAL_ADDRESS;

Description

The `AllocatePages()` function allocates the requested number of pages and returns a pointer to the base address of the page range in the location referenced by `Memory`. The function scans the memory map to locate free pages. When it finds a physically contiguous block of pages that is large enough and also satisfies the allocation requirements of `Type`, it changes the memory map to indicate that the pages are now of type `MemoryType`.

In general, UEFI OS loaders and UEFI applications should allocate memory (and pool) of type `EfiLoaderData`. UEFI boot service drivers must allocate memory (and pool) of type `EfiBootServicesData`. UEFI runtime drivers should allocate memory (and pool) of type `EfiRuntimeServicesData` (although such allocation can only be made during boot services time).

Allocation requests of `Type AllocateAnyPages` allocate any available range of pages that satisfies the request. On input, the address pointed to by `Memory` is ignored.

Allocation requests of `Type AllocateMaxAddress` allocate any available range of pages whose uppermost address is less than or equal to the address pointed to by `Memory` on input.

Allocation requests of `Type AllocateAddress` allocate pages at the address pointed to by `Memory` on input.

**Note:** UEFI drivers and UEFI applications that are not targeted for a specific implementation must perform memory allocations for the following runtime types using `AllocateAnyPages` address mode:

- `EfiACPIReclaimMemory`,
EfiACPIMemoryNVS,
EfiRuntimeServicesCode,
EfiRuntimeServicesData,
EfiReservedMemoryType.

Status Codes Returned

<table>
<thead>
<tr>
<th>Status Code</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>EFI_SUCCESS</td>
<td>The requested pages were allocated.</td>
</tr>
<tr>
<td>EFI_OUT_OF_RESOURCES</td>
<td>The pages could not be allocated.</td>
</tr>
<tr>
<td>EFI_INVALID_PARAMETER</td>
<td>Type is not AllocateAnyPages or AllocateMaxAddress or AllocateAddress.</td>
</tr>
<tr>
<td>EFI_INVALID_PARAMETER</td>
<td>MemoryType is in the range EfiMaxMemoryType..0x6FFFFFF.</td>
</tr>
<tr>
<td>EFI_INVALID_PARAMETER</td>
<td>MemoryType is EfiPersistentMemory or EfiUnacceptedMemoryType.</td>
</tr>
<tr>
<td>EFI_INVALID_PARAMETER</td>
<td>Memory is NULL.</td>
</tr>
<tr>
<td>EFI_NOT_FOUND</td>
<td>The requested pages could not be found.</td>
</tr>
</tbody>
</table>

EFI_BOOT_SERVICES.FreePages()

Summary
Frees memory pages.

Prototype
typedef
    EFI_STATUS
    (EFIAPI *EFI_FREE_PAGES) (  
    IN EFI_PHYSICAL_ADDRESS Memory,
    IN UINTN        Pages);

Parameters
Memory The base physical address of the pages to be freed. Type
EFI_PHYSICAL_ADDRESS is defined in the
EFI_BOOT_SERVICES.AllocatePages() function description.
Pages The number of contiguous 4 KiB pages to free.

Description
The FreePages() function returns memory allocated by AllocatePages() to the firmware.
Status Codes Returned

<table>
<thead>
<tr>
<th>Code</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>EFI_SUCCESS</td>
<td>The requested memory pages were freed.</td>
</tr>
<tr>
<td>EFI_NOT_FOUND</td>
<td>The requested memory pages were not allocated with AllocatePages().</td>
</tr>
<tr>
<td>EFI_INVALID_PARAMETER</td>
<td>Memory is not a page-aligned address or Pages is invalid.</td>
</tr>
</tbody>
</table>

EFI_BOOT_SVCES.GetMemoryMap()

Summary
Returns the current memory map.

Prototype
```c
typedef EFI_STATUS
  (EFIAPI *EFI_GET_MEMORY_MAP) (  
    IN OUT UINTN *MemoryMapSize,
    OUT EFI_MEMORY_DESCRIPTOR *MemoryMap,
    OUT UINTN *MapKey,
    OUT UINTN *DescriptorSize,
    OUT UINT32 *DescriptorVersion
  );
```

Parameters

- **MemoryMapSize** A pointer to the size, in bytes, of the MemoryMap buffer. On input, this is the size of the buffer allocated by the caller. On output, it is the size of the buffer returned by the firmware if the buffer was large enough, or the size of the buffer needed to contain the map if the buffer was too small.
- **MemoryMap** A pointer to the buffer in which firmware places the current memory map. The map is an array of EFI_MEMORY_DESCRIPTORs. See “Related Definitions.”
- **MapKey** A pointer to the location in which firmware returns the key for the current memory map.
- **DescriptorSize** A pointer to the location in which firmware returns the size, in bytes, of an individual EFI_MEMORY_DESCRIPTOR.
- **DescriptorVersion** A pointer to the location in which firmware returns the version number associated with the EFI_MEMORY_DESCRIPTOR. See “Related Definitions.”
Related Definitions

```c
typedef struct {
    UINT32 Type;
    EFI_PHYSICAL_ADDRESS PhysicalStart;
    EFI_VIRTUAL_ADDRESS VirtualStart;
    UINT64 NumberOfPages;
    UINT64 Attribute;
} EFI_MEMORY_DESCRIPTOR;
```

- **Type**
  Type of the memory region. Type `EFI_MEMORY_TYPE` is defined in the `AllocatePages()` function description.

- **PhysicalStart**
  Physical address of the first byte in the memory region. `PhysicalStart` must be aligned on a 4 KiB boundary, and must not be above `0xfffffffffffff000`. Type `EFI_PHYSICAL_ADDRESS` is defined in the `AllocatePages()` function description.

- **VirtualStart**
  Virtual address of the first byte in the memory region. `VirtualStart` must be aligned on a 4 KiB boundary, and must not be above `0xfffffffffffff000`. Type `EFI_VIRTUAL_ADDRESS` is defined in “Related Definitions.”

- **NumberOfPages**
  Number of 4 KiB pages in the memory region. `NumberOfPages` must not be 0, and must not be any value that would represent a memory page with a start address, either physical or virtual, above `0xfffffffffffff000`.

- **Attribute**
  Attributes of the memory region that describe the bit mask of capabilities for that memory region, and not necessarily the current settings for that memory region. See the following “Memory Attribute Definitions.”
Memory Attribute Definitions

These types can be “ORed” together as needed:

```c
#define EFI_MEMORY_UC            0x0000000000000001
#define EFI_MEMORY_WC            0x0000000000000002
#define EFI_MEMORY_WT            0x0000000000000004
#define EFI_MEMORY_WB            0x0000000000000008
#define EFI_MEMORY_UCE           0x0000000000000010
#define EFI_MEMORY_WP            0x0000000000001000
#define EFI_MEMORY_RP            0x0000000000002000
#define EFI_MEMORY_XP            0x0000000000004000
#define EFI_MEMORY_NV            0x0000000000008000
#define EFI_MEMORY_MORE_RELIABLE 0x0000000000010000
#define EFI_MEMORY_RO            0x0000000000020000
#define EFI_MEMORY_SP            0x0000000000040000
#define EFI_MEMORY_CPU_CRYPTO    0x0000000000080000
#define EFI_MEMORY_RUNTIME       0x8000000000000000
```

- **EFI_MEMORY_UC**: Memory cacheability attribute: The memory region supports being configured as not cacheable.
- **EFI_MEMORY_WC**: Memory cacheability attribute: The memory region supports being configured as write combining.
- **EFI_MEMORY_WT**: Memory cacheability attribute: The memory region supports being configured as cacheable with a “write through” policy. Writes that hit in the cache will also be written to main memory.
- **EFI_MEMORY_WB**: Memory cacheability attribute: The memory region supports being configured as cacheable with a “write back” policy. Reads and writes that hit in the cache do not propagate to main memory. Dirty data is written back to main memory when a new cache line is allocated.
- **EFI_MEMORY_UCE**: Memory cacheability attribute: The memory region supports being configured as not cacheable, exported, and supports the “fetch and add” semaphore mechanism.
- **EFI_MEMORY_WP**: Physical memory protection attribute: The memory region supports being configured as write-protected by system hardware. This is typically used as a cacheability attribute today. The memory region supports being configured as cacheable with a "write protected" policy. Reads come from cache lines when possible, and read misses cause cache fills. Writes are propagated to the system bus and cause corresponding cache lines on all processors on the bus to be invalidated.
- **EFI_MEMORY_SP**: Specific-purpose memory (SPM). The memory is earmarked for specific purposes such as for specific device drivers or applications. The SPM attribute serves as a hint to the OS to avoid allocating this memory for core OS data or code that can not be relocated. Prolonged use of this memory for purposes other than the intended purpose may result in suboptimal platform performance.
If this flag is set, the memory region is capable of being protected with the CPU’s memory cryptographic capabilities. If this flag is clear, the memory region is not capable of being protected with the CPU’s memory cryptographic capabilities or the CPU does not support CPU memory cryptographic capabilities.

**Note:** UEFI spec 2.5 and following: use EFI_MEMORY_RO as write-protected physical memory protection attribute. Also, EFI_MEMORY_WP means cacheability attribute.

**EFI_MEMORY_RP**  
Physical memory protection attribute: The memory region supports being configured as read-protected by system hardware.

**EFI_MEMORY_XP**  
Physical memory protection attribute: The memory region supports being configured so it is protected by system hardware from executing code.

**EFI_MEMORY_NV**  
Runtime memory attribute: The memory region refers to persistent memory

**EFI_MEMORY_MORE_RELIABLE**  
The memory region provides higher reliability relative to other memory in the system. If all memory has the same reliability, then this bit is not used.

**EFI_MEMORY_RO**  
Physical memory protection attribute: The memory region supports making this memory range read-only by system hardware.

**EFI_MEMORY_RUNTIME**  
Runtime memory attribute: The memory region needs to be given a virtual mapping by the operating system when `SetVirtualAddressMap()` is called (described in Section 8.4).

```c
//*******************************************************
//EFI_VIRTUAL_ADDRESS
//*******************************************************
typedef UINT64  EFI_VIRTUAL_ADDRESS;

//******************************************************************************
// Memory Descriptor Version Number
//******************************************************************************
#define EFI_MEMORY_DESCRIPTOR_VERSION 1
```

**Description**

The `GetMemoryMap()` function returns a copy of the current memory map. The map is an array of memory descriptors, each of which describes a contiguous block of memory. The map describes all of memory, no matter how it is being used. That is, it includes blocks allocated by `EFI_BOOT_SERVICES.AllocatePages()` and `EFI_BOOT_SERVICES.AllocatePool()`, as well as blocks that the firmware is using for its own purposes. The memory map is only used to describe memory that is present in the system. The firmware does not return a range description for address space regions that are not backed by physical hardware. Regions that are backed by physical hardware, but are not supposed to be accessed by the OS, must be returned as `EfiReservedMemoryType`. The OS may use addresses of memory ranges that are not described in the memory map at its own discretion.
Until **EFI_BOOT_SERVICES.ExitBootServices()** is called, the memory map is owned by the firmware and the currently executing UEFI Image should only use memory pages it has explicitly allocated.

If the **MemoryMap** buffer is too small, the **EFI_BUFFER_TOO_SMALL** error code is returned and the **MemoryMapSize** value contains the size of the buffer needed to contain the current memory map. The actual size of the buffer allocated for the consequent call to **GetMemoryMap()** should be bigger then the value returned in **MemoryMapSize**, since allocation of the new buffer may potentially increase memory map size.

On success a **MapKey** is returned that identifies the current memory map. The firmware’s key is changed every time something in the memory map changes. In order to successfully invoke **EFI_BOOT_SERVICES.ExitBootServices()** the caller must provide the current memory map key.

The **GetMemoryMap()** function also returns the size and revision number of the **EFI_MEMORY_DESCRIPTOR**. The **DescriptorSize** represents the size in bytes of an **EFI_MEMORY_DESCRIPTOR** array element returned in **MemoryMap**. The size is returned to allow for future expansion of the **EFI_MEMORY_DESCRIPTOR** in response to hardware innovation. The structure of the **EFI_MEMORY_DESCRIPTOR** may be extended in the future but it will remain backwards compatible with the current definition. Thus OS software must use the **DescriptorSize** to find the start of each **EFI_MEMORY_DESCRIPTOR** in the **MemoryMap** array.

### Status Codes Returned

<table>
<thead>
<tr>
<th>Status Code</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>EFI_SUCCESS</strong></td>
<td>The memory map was returned in the <strong>MemoryMap</strong> buffer.</td>
</tr>
<tr>
<td><strong>EFI_BUFFER_TOO_SMALL</strong></td>
<td>The <strong>MemoryMap</strong> buffer was too small. The current buffer size needed to</td>
</tr>
<tr>
<td></td>
<td>hold the memory map is returned in <strong>MemoryMapSize</strong>.</td>
</tr>
<tr>
<td><strong>EFI_INVALID_PARAMETER</strong></td>
<td><strong>MemoryMapSize</strong> is <strong>NULL</strong>.</td>
</tr>
<tr>
<td><strong>EFI_INVALID_PARAMETER</strong></td>
<td>The <strong>MemoryMap</strong> buffer is not too small and <strong>MemoryMap</strong> is <strong>NULL</strong>.</td>
</tr>
</tbody>
</table>

**EFI_BOOT_SERVICES.AllocatePool()**

**Summary**

Allocates pool memory.

**Prototype**

```c
typedef
EFI_STATUS
(EFIAPI *EFI_ALLOCATE_POOL) (  
    IN EFI_MEMORY_TYPE PoolType,
    IN UINTN Size,
    OUT VOID **Buffer
);
```

**Parameters**

- **PoolType** The type of pool to allocate. Type **EFI_MEMORY_TYPE** is defined in the **EFI_BOOT_SERVICES.AllocatePages()** function description. **PoolType** values in the range 0x70000000..0x7FFFFFFF
PoolType values in the range 0x80000000..0xFFFFFFFF are reserved for use by UEFI OS loaders that are provided by operating system vendors.

<table>
<thead>
<tr>
<th>Size</th>
<th>The number of bytes to allocate from the pool.</th>
</tr>
</thead>
<tbody>
<tr>
<td>Buffer</td>
<td>A pointer to a pointer to the allocated buffer if the call succeeds; undefined otherwise.</td>
</tr>
</tbody>
</table>

**Note:** UEFI applications and UEFI drivers must not allocate memory of type EfiReservedMemoryType.

### Description

The `AllocatePool()` function allocates a memory region of Size bytes from memory of type PoolType and returns the address of the allocated memory in the location referenced by Buffer. This function allocates pages from EfiConventionalMemory as needed to grow the requested pool type. All allocations are eight-byte aligned.

The allocated pool memory is returned to the available pool with the EFI_BOOT_SERVICES.FreePool() function.

**Status Codes Returned**

<table>
<thead>
<tr>
<th>Status Code</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>EFI_SUCCESS</td>
<td>The requested number of bytes was allocated.</td>
</tr>
<tr>
<td>EFI_OUT_OF_RESOURCES</td>
<td>The pool requested could not be allocated.</td>
</tr>
<tr>
<td>EFI_INVALID_PARAMETER</td>
<td>PoolType is in the range EfiMaxMemoryType..0xFFFFFFF.</td>
</tr>
<tr>
<td>EFI_INVALID_PARAMETER</td>
<td>PoolType is EfiPersistentMemory.</td>
</tr>
<tr>
<td>EFI_INVALID_PARAMETER</td>
<td>Buffer is NULL.</td>
</tr>
</tbody>
</table>

### EFI_BOOT_SERVICES.FreePool()

**Summary**

Returns pool memory to the system.

**Prototype**

```c
typedef EFI_STATUS
    (EFIAPI *EFI_FREE_POOL) (IN VOID *Buffer);
```

**Parameters**

- **Buffer**: Pointer to the buffer to free.

**Description**

The `FreePool()` function returns the memory specified by Buffer to the system. On return, the memory’s type is EfiConventionalMemory. The Buffer that is freed must have been allocated by `AllocatePool()`.
Status Codes Returned

<table>
<thead>
<tr>
<th>Code</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>EFI_SUCCESS</td>
<td>The memory was returned to the system.</td>
</tr>
<tr>
<td>EFI_INVALID_PARAMETER</td>
<td>Buffer was invalid.</td>
</tr>
</tbody>
</table>

7.3 Protocol Handler Services

In the abstract, a protocol consists of a 128-bit globally unique identifier (GUID) and a Protocol Interface structure. The structure contains the functions and instance data that are used to access a device. The functions that make up Protocol Handler Services allow applications to install a protocol on a handle, identify the handles that support a given protocol, determine whether a handle supports a given protocol, and so forth. See Table 7-7.

Table 7-7 Protocol Interface Functions

<table>
<thead>
<tr>
<th>Name</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>InstallProtocolInterface</td>
<td>Boot</td>
<td>Installs a protocol interface on a device handle.</td>
</tr>
<tr>
<td>UninstallProtocolInterface</td>
<td>Boot</td>
<td>Removes a protocol interface from a device handle.</td>
</tr>
<tr>
<td>ReinstallProtocolInterface</td>
<td>Boot</td>
<td>Reinstalls a protocol interface on a device handle.</td>
</tr>
<tr>
<td>RegisterProtocolNotify</td>
<td>Boot</td>
<td>Registers an event that is to be signaled whenever an interface is installed for a specified protocol.</td>
</tr>
<tr>
<td>LocateHandle</td>
<td>Boot</td>
<td>Returns an array of handles that support a specified protocol.</td>
</tr>
<tr>
<td>HandleProtocol</td>
<td>Boot</td>
<td>Queries a handle to determine if it supports a specified protocol.</td>
</tr>
<tr>
<td>LocateDevicePath</td>
<td>Boot</td>
<td>Locates all devices on a device path that support a specified protocol and returns the handle to the device that is closest to the path.</td>
</tr>
<tr>
<td>OpenProtocol</td>
<td>Boot</td>
<td>Adds elements to the list of agents consuming a protocol interface.</td>
</tr>
<tr>
<td>CloseProtocol</td>
<td>Boot</td>
<td>Removes elements from the list of agents consuming a protocol interface.</td>
</tr>
<tr>
<td>OpenProtocolInformation</td>
<td>Boot</td>
<td>Retrieve the list of agents that are currently consuming a protocol interface.</td>
</tr>
<tr>
<td>ConnectController</td>
<td>Boot</td>
<td>Uses a set of precedence rules to find the best set of drivers to manage a controller.</td>
</tr>
<tr>
<td>DisconnectController</td>
<td>Boot</td>
<td>Informs a set of drivers to stop managing a controller.</td>
</tr>
<tr>
<td>ProtocolsPerHandle</td>
<td>Boot</td>
<td>Retrieves the list of protocols installed on a handle. The return buffer is automatically allocated.</td>
</tr>
<tr>
<td>LocateHandleBuffer</td>
<td>Boot</td>
<td>Retrieves the list of handles from the handle database that meet the search criteria. The return buffer is automatically allocated.</td>
</tr>
<tr>
<td>LocateProtocol</td>
<td>Boot</td>
<td>Finds the first handle in the handle database the supports the requested protocol.</td>
</tr>
<tr>
<td>InstallMultipleProtocolInterfaces</td>
<td>Boot</td>
<td>Installs one or more protocol interfaces onto a handle.</td>
</tr>
<tr>
<td>UninstallMultipleProtocolInterfaces</td>
<td>Boot</td>
<td>Uninstalls one or more protocol interfaces from a handle.</td>
</tr>
</tbody>
</table>
The Protocol Handler boot services have been modified to take advantage of the information that is now being tracked with the `EFI_BOOT_SERVICES.OpenProtocol()` and `EFI_BOOT_SERVICES.CloseProtocol()` boot services. Since the usage of protocol interfaces is being tracked with these new boot services, it is now possible to safely uninstall and reinstall protocol interfaces that are being consumed by UEFI drivers.

As depicted in Figure 7-1, the firmware is responsible for maintaining a “data base” that shows which protocols are attached to each device handle. (The figure depicts the “data base” as a linked list, but the choice of data structure is implementation-dependent.) The “data base” is built dynamically by calling the `EFI_BOOT_SERVICES.InstallProtocolInterface()` function. Protocols can only be installed by UEFI drivers or the firmware itself. In the figure, a device handle (`EFI_HANDLE`) refers to a list of one or more registered protocol interfaces for that handle. The first handle in the system has four attached protocols, and the second handle has two attached protocols. Each attached protocol is represented as a GUID/Interface pointer pair. The GUID is the name of the protocol, and Interface points to a protocol instance. This data structure will typically contain a list of interface functions, and some amount of instance data.

Access to devices is initiated by calling the `EFI_BOOT_SERVICES.HandleProtocol()` function, which determines whether a handle supports a given protocol. If it does, a pointer to the matching Protocol Interface structure is returned.

When a protocol is added to the system, it may either be added to an existing device handle or it may be added to create a new device handle. Figure 7-1 shows that protocol handlers are listed for each device handle and that each protocol handler is logically a UEFI driver.
The ability to add new protocol interfaces as new handles or to layer them on existing interfaces provides
great flexibility. Layering makes it possible to add a new protocol that builds on a device’s basic protocols. An example of this might be to layer on a EFI_SIMPLE_TEXT_OUTPUT_PROTOCOL support that would build on the handle’s underlying EFI_SERIAL_IO_PROTOCOL.

The ability to add new handles can be used to generate new devices as they are found, or even to generate abstract devices. An example of this might be to add a multiplexing device that replaces ConsoleOut with a virtual device that multiplexes the EFI_SIMPLE_TEXT_OUTPUT_PROTOCOL protocol onto multiple underlying device handles.

**Driver Model Boot Services**

Following is a detailed description of the new UEFI boot services that are required by the UEFI Driver Model. These boot services are being added to reduce the size and complexity of the bus drivers and device drivers. This, in turn, will reduce the amount of ROM space required by drivers that are programmed into ROMs on adapters or into system FLASH, and reduce the development and testing time required by driver writers.

These new services fall into two categories. The first group is used to track the usage of protocol interfaces by different agents in the system. Protocol interfaces are stored in a handle database. The handle database consists of a list of handles, and on each handle there is a list of one or more protocol interfaces. The boot services EFI_BOOT_SERVICES.InstallProtocolInterface(), EFI_BOOT_SERVICES.UninstallProtocolInterface(), and
**EFI_BOOT_SERVICES.ReinstallProtocolInterface()** are used to add, remove, and replace protocol interfaces in the handle database. The boot service **EFI_BOOT_SERVICES.HandleProtocol()** is used to look up a protocol interface in the handle database. However, agents that call **HandleProtocol()** are not tracked, so it is not safe to call **UninstallProtocolInterface()** or **ReinstallProtocolInterface()** because an agent may be using the protocol interface that is being removed or replaced.

The solution is to track the usage of protocol interfaces in the handle database itself. To accomplish this, each protocol interface includes a list of agents that are consuming the protocol interface. **Figure 7-2** shows an example handle database with these new agent lists. An agent consists of an image handle, a controller handle, and some attributes. The image handle identifies the driver or application that is consuming the protocol interface. The controller handle identifies the controller that is consuming the protocol interface. Since a driver may manage more than one controller, the combination of a driver's image handle and a controller's controller handle uniquely identifies the agent that is consuming the protocol interface. The attributes show how the protocol interface is being used.

![Figure 7-2 Handle Database](OM13156)

In order to maintain these agent lists in the handle database, some new boot services are required. These are **EFI_BOOT_SERVICES.OpenProtocol()**, **EFI_BOOT_SERVICES.CloseProtocol()**, and **EFI_BOOT_SERVICES.OpenProtocolInformation()**. **OpenProtocol()** adds elements to the list of agents consuming a protocol interface. **CloseProtocol()** removes elements from the list of agents.
consuming a protocol interface, and `EFI_BOOT_SERVICES.OpenProtocolInformation()` retrieves the entire list of agents that are currently using a protocol interface.

The second group of boot services is used to deterministically connect and disconnect drivers to controllers. The boot services in this group are `EFI_BOOT_SERVICES.ConnectController()` and `EFI_BOOT_SERVICES.DisconnectController()`. These services take advantage of the new features of the handle database along with the new protocols described in this document to manage the drivers and controllers present in the system. `ConnectController()` uses a set of strict precedence rules to find the best set of drivers for a controller. This provides a deterministic matching of drivers to controllers with extensibility mechanisms for OEMs, IBVs, and IHVs. `DisconnectController()` allows drivers to be disconnected from controllers in a controlled manner, and by using the new features of the handle database it is possible to fail a disconnect request because a protocol interface cannot be released at the time of the disconnect request.

The third group of boot services is designed to help simplify the implementation of drivers, and produce drivers with smaller executable footprints. The `EFI_BOOT_SERVICES.LocateHandleBuffer()` is a new version of `EFI_BOOT_SERVICES.LocateHandle()` that allocates the required buffer for the caller. This eliminates two calls to `LocateHandle()` and a call to ` EFI_BOOT_SERVICES.AllocatePool()` from the caller's code. `EFI_BOOT_SERVICES.LocateProtocol()` searches the handle database for the first protocol instance that matches the search criteria. The `EFI_BOOT_SERVICES.InstallMultipleProtocolInterfaces()` and `EFI_BOOT_SERVICES.UninstallMultipleProtocolInterfaces()` are very useful to driver writers. These boot services allow one or more protocol interfaces to be added or removed from a handle. In addition, ` EFI_BOOT_SERVICES.InstallMultipleProtocolInterfaces()` guarantees that a duplicate device path is never added to the handle database. This is very useful to bus drivers that can create one child handle at a time, because it guarantees that the bus driver will not inadvertently create two instances of the same child handle.

**EFI_BOOT_SERVICES.InstallProtocolInterface()**

**Summary**

Installs a protocol interface on a device handle. If the handle does not exist, it is created and added to the list of handles in the system. ` EFI_BOOT_SERVICES.InstallMultipleProtocolInterfaces()` performs more error checking than ` EFI_BOOT_SERVICES.InstallProtocolInterface()`, so it is recommended that `EFI_BOOT_SERVICES.InstallMultipleProtocolInterfaces()` be used in place of ` EFI_BOOT_SERVICES.InstallProtocolInterface()`.

**Prototype**
typedef
EFI_STATUS
(EFI_API *EFI_INSTALL_PROTOCOL_INTERFACE) (
    IN OUT EFI_HANDLE *Handle,
    IN EFI_GUID *Protocol,
    IN EFI_INTERFACE_TYPE InterfaceType,
    IN VOID *Interface
);
typedef VOID *EFI_HANDLE;

typedef struct {
    UINT32 Data1;
    UINT16 Data2;
    UINT16 Data3;
    UINT8 Data4[8];
} EFI_GUID;

typedef enum {
    EFI_NATIVE_INTERFACE
} EFI_INTERFACE_TYPE;

Description
The InstallProtocolInterface() function installs a protocol interface (a GUID/Protocol Interface structure pair) on a device handle. The same GUID cannot be installed more than once onto the same handle. If installation of a duplicate GUID on a handle is attempted, an EFI_INVALID_PARAMETER will result.

Installing a protocol interface allows other components to locate the Handle, and the interfaces installed on it.

When a protocol interface is installed, the firmware calls all notification functions that have registered to wait for the installation of Protocol. For more information, see the EFI_BOOT_SERVICES.RegisterProtocolNotify() function description.
Status Codes Returned

<table>
<thead>
<tr>
<th>Code</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>EFI_SUCCESS</td>
<td>The protocol interface was installed.</td>
</tr>
<tr>
<td>EFI_OUT_OF_RESOURCES</td>
<td>Space for a new handle could not be allocated.</td>
</tr>
<tr>
<td>EFI_INVALID_PARAMETER</td>
<td>Handle is <strong>NULL</strong></td>
</tr>
<tr>
<td>EFI_INVALID_PARAMETER</td>
<td>Protocol is <strong>NULL</strong></td>
</tr>
<tr>
<td>EFI_INVALID_PARAMETER</td>
<td>InterfaceType is not <strong>EFI_NATIVE_INTERFACE</strong></td>
</tr>
<tr>
<td>EFI_INVALID_PARAMETER</td>
<td>Protocol is already installed on the handle specified by Handle.</td>
</tr>
</tbody>
</table>

**EFI_BOOT_SERVICES.UninstallProtocolInterface()**

**Summary**
Removes a protocol interface from a device handle. It is recommended that **UninstallMultipleProtocolInterfaces()** be used in place of **UninstallProtocolInterface().**

**Prototype**

```c
typedef EFI_STATUS
  (EFIAPICALLTYPE *EFI_UNINSTALL_PROTOCOL_INTERFACE) ( 
    IN EFI_HANDLE Handle,
    IN EFI_GUID *Protocol,
    IN VOID *Interface
  );
```

**Parameters**

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Handle</td>
<td>The handle on which the interface was installed. If <strong>Handle</strong> is not a valid handle, then <strong>EFI_INVALID_PARAMETER</strong> is returned. Type <strong>EFI_HANDLE</strong> is defined in the <strong>EFI_BOOT_SERVICES.InstallProtocolInterface()</strong> function description.</td>
</tr>
<tr>
<td>Protocol</td>
<td>The numeric ID of the interface. It is the caller’s responsibility to pass in a valid GUID. See “Wired For Management Baseline” for a description of valid GUID values. Type <strong>EFI_GUID</strong> is defined in the <strong>InstallProtocolInterface()</strong> function description.</td>
</tr>
<tr>
<td>Interface</td>
<td>A pointer to the interface. <strong>NULL</strong> can be used if a structure is not associated with <strong>Protocol.</strong></td>
</tr>
</tbody>
</table>

**Description**

The **UninstallProtocolInterface()** function removes a protocol interface from the handle on which it was previously installed. The **Protocol** and **Interface** values define the protocol interface to remove from the handle.

The caller is responsible for ensuring that there are no references to a protocol interface that has been removed. In some cases, outstanding reference information is not available in the protocol, so the
protocol, once added, cannot be removed. Examples include Console I/O, Block I/O, Disk I/O, and (in general) handles to device protocols.

If the last protocol interface is removed from a handle, the handle is freed and is no longer valid.

**EFI 1.10 Extension**

The extension to this service directly addresses the limitations described in the section above. There may be some drivers that are currently consuming the protocol interface that needs to be uninstalled, so it may be dangerous to just blindly remove a protocol interface from the system. Since the usage of protocol interfaces is now being tracked for components that use the `EFI_BOOT_SERVICES.OpenProtocol()` and `EFI_BOOT_SERVICES.CloseProtocol()` boot services, a safe version of this function can be implemented. Before the protocol interface is removed, an attempt is made to force all the drivers that are consuming the protocol interface to stop consuming that protocol interface. This is done by calling the boot service `EFI_BOOT_SERVICES.DisconnectController()` for the driver that currently have the protocol interface open with an attribute of `EFI_OPEN_PROTOCOL_BY_DRIVER` or `EFI_OPEN_PROTOCOL_BY_DRIVER | EFI_OPEN_PROTOCOL_EXCLUSIVE`.

If the disconnect succeeds, then those agents will have called the boot service `EFI_BOOT_SERVICES.CloseProtocol()` to release the protocol interface. Lastly, all of the agents that have the protocol interface open with an attribute of `EFI_OPEN_PROTOCOL_BY_HANDLE_PROTOCOL`, `EFI_OPEN_PROTOCOL_GET_PROTOCOL`, or `EFI_OPEN_PROTOCOL_TEST_PROTOCOL` are closed. If there are any agents remaining that still have the protocol interface open, the protocol interface is not removed from the handle and `EFI_ACCESS_DENIED` is returned. In addition, all of the drivers that were disconnected with the boot service `DisconnectController()` earlier, are reconnected with the boot service `EFI_BOOT_SERVICES.ConnectController()`. If there are no agents remaining that are consuming the protocol interface, then the protocol interface is removed from the handle as described above.

**Status Codes Returned**

<table>
<thead>
<tr>
<th>Code</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>EFI_SUCCESS</td>
<td>The interface was removed.</td>
</tr>
<tr>
<td>EFI_NOT_FOUND</td>
<td>The interface was not found.</td>
</tr>
<tr>
<td>EFI_ACCESS_DENIED</td>
<td>The interface was not removed because the interface is still being used by a driver.</td>
</tr>
<tr>
<td>EFI_INVALID_PARAMETER</td>
<td>Handle is <strong>NULL</strong>.</td>
</tr>
<tr>
<td>EFI_INVALID_PARAMETER</td>
<td>Protocol is <strong>NULL</strong>.</td>
</tr>
</tbody>
</table>

**EFI_BOOT_SERVICES.ReinstallProtocolInterface()**

**Summary**

Reinstalls a protocol interface on a device handle.
Prototype

```c
typedef
EFI_STATUS
(EFIAPI *EFI_REINSTALL_PROTOCOL_INTERFACE) (
  IN EFI_HANDLE Handle,
  IN EFI_GUID *Protocol,
  IN VOID *OldInterface,
  IN VOID *NewInterface
);
```

Parameters

- **Handle**
  Handle on which the interface is to be reinstalled. If `Handle` is not a valid handle, then `EFI_INVALID_PARAMETER` is returned. Type `EFI_HANDLE` is defined in the `EFI_BOOT_SERVICES.InstallProtocolInterface()` function description.

- **Protocol**
  The numeric ID of the interface. It is the caller’s responsibility to pass in a valid GUID. See “Wired For Management Baseline” for a description of valid GUID values. Type `EFI_GUID` is defined in the `InstallProtocolInterface()` function description.

- **OldInterface**
  A pointer to the old interface. `NULL` can be used if a structure is not associated with `Protocol`.

- **NewInterface**
  A pointer to the new interface. `NULL` can be used if a structure is not associated with `Protocol`.

Description

The `ReinstallProtocolInterface()` function reinstalls a protocol interface on a device handle. The `OldInterface` for `Protocol` is replaced by the `NewInterface`. `NewInterface` may be the same as `OldInterface`. If it is, the registered protocol notifies occur for the handle without replacing the interface on the handle.

As with `InstallProtocolInterface()`, any process that has registered to wait for the installation of the interface is notified.

The caller is responsible for ensuring that there are no references to the `OldInterface` that is being removed.

**EFI 1.10 Extension**

The extension to this service directly addresses the limitations described in the section above. There may be some number of drivers currently consuming the protocol interface that is being reinstalled. In this case, it may be dangerous to replace a protocol interface in the system. It could result in an unstable state, because a driver may attempt to use the old protocol interface after a new one has been reinstalled. Since the usage of protocol interfaces is now being tracked for components that use the `EFI_BOOT_SERVICES.OpenProtocol()` and `EFI_BOOT_SERVICES.CloseProtocol()` boot services, a safe version of this function can be implemented.

When this function is called, a call is first made to the boot service `UninstallProtocolInterface()`. This will guarantee that all of the agents are currently consuming the protocol interface `OldInterface` will stop using `OldInterface`. If `UninstallProtocolInterface()` returns `EFI_ACCESS_DENIED`, then
this function returns \texttt{EFI\_ACCESS\_DENIED}, \texttt{OldInterface} remains on \texttt{Handle}, and the protocol notifies are not processed because \texttt{NewInterface} was never installed.

If \texttt{UninstallProtocolInterface()} succeeds, then a call is made to the boot service \texttt{EFI\_BOOT\_SERVICES.InstallProtocolInterface()} to put the \texttt{NewInterface} onto \texttt{Handle}.

Finally, the boot service \texttt{EFI\_BOOT\_SERVICES.ConnectController()} is called so all agents that were forced to release \texttt{OldInterface} with \texttt{UninstallProtocolInterface()} can now consume the protocol interface \texttt{NewInterface} that was installed with \texttt{InstallProtocolInterface()}. After \texttt{OldInterface} has been replaced with \texttt{NewInterface}, any process that has registered to wait for the installation of the interface is notified.

### Status Codes Returned

<table>
<thead>
<tr>
<th>Status Code</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>\texttt{EFI_SUCCESS}</td>
<td>The protocol interface was reinstalled.</td>
</tr>
<tr>
<td>\texttt{EFI_NOT_FOUND}</td>
<td>The \texttt{OldInterface} on the handle was not found.</td>
</tr>
<tr>
<td>\texttt{EFI_ACCESS_DENIED}</td>
<td>The protocol interface could not be reinstalled, because \texttt{OldInterface} is still being used by a driver that will not release it.</td>
</tr>
<tr>
<td>\texttt{EFI_INVALID_PARAMETER}</td>
<td>\texttt{Handle} is \texttt{NULL}.</td>
</tr>
<tr>
<td>\texttt{EFI_INVALID_PARAMETER}</td>
<td>\texttt{Protocol} is \texttt{NULL}.</td>
</tr>
</tbody>
</table>

### \texttt{EFI\_BOOT\_SERVICES.RegisterProtocolNotify()}

#### Summary

Creates an event that is to be signaled whenever an interface is installed for a specified protocol.

#### Prototype

```c
typedef EFI_STATUS (EFIAPI *EFI_REGISTER_PROTOCOL_NOTIFY) (
    IN EFI_GUID *Protocol,
    IN EFI_EVENT Event,
    OUT VOID **Registration
);
```

#### Parameters

- **Protocol**
  
  The numeric ID of the protocol for which the event is to be registered. Type \texttt{EFI\_GUID} is defined in the \texttt{EFI\_BOOT\_SERVICES.InstallProtocolInterface()} function description.

- **Event**
  
  Event that is to be signaled whenever a protocol interface is registered for \texttt{Protocol}. The type \texttt{EFI\_EVENT} is defined in the \texttt{CreateEvent()} function description. The same \texttt{EFI\_EVENT} may be used for multiple protocol notify registrations.

- **Registration**
  
  A pointer to a memory location to receive the registration value. This value must be saved and used by the notification function of \texttt{Event}.
to retrieve the list of handles that have added a protocol interface of type Protocol.

Description

The **RegisterProtocolNotify()** function creates an event that is to be signaled whenever a protocol interface is installed for Protocol by **InstallProtocolInterface()** or **EFI_BOOT_SERVICES.ReinstallProtocolInterface()**.

Once Event has been signaled, the **EFI_BOOT_SERVICES.LocateHandle()** function can be called to identify the newly installed, or reinstalled, handles that support Protocol. The **Registration** parameter in **EFI_BOOT_SERVICES.RegisterProtocolNotify()** corresponds to the **SearchKey** parameter in **LocateHandle()**. Note that the same handle may be returned multiple times if the handle reinstalls the target protocol ID multiple times. This is typical for removable media devices, because when such a device reappears, it will reinstall the Block I/O protocol to indicate that the device needs to be checked again. In response, layered Disk I/O and Simple File System protocols may then reinstall their protocols to indicate that they can be re-checked, and so forth.

Events that have been registered for protocol interface notification can be unregistered by calling **CloseEvent()**.

Status Codes Returned

<table>
<thead>
<tr>
<th>Status Code</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>EFI_SUCCESS</td>
<td>The notification event has been registered.</td>
</tr>
<tr>
<td>EFI_OUT_OF_RESOURCES</td>
<td>Space for the notification event could not be allocated.</td>
</tr>
<tr>
<td>EFI_INVALID_PARAMETER</td>
<td>Protocol is NULL.</td>
</tr>
<tr>
<td>EFI_INVALID_PARAMETER</td>
<td>Event is NULL.</td>
</tr>
<tr>
<td>EFI_INVALID_PARAMETER</td>
<td>Registration is NULL.</td>
</tr>
</tbody>
</table>

**EFI_BOOT_SERVICES.LocateHandle()**

**Summary**

Returns an array of handles that support a specified protocol.

**Prototype**

```c
typedef
    EFI_STATUS
    (EFIAPI *EFI_LOCATE_HANDLE) (
    IN EFI_LOCATE_SEARCH_TYPE SearchType,
    IN EFI_GUID *Protocol OPTIONAL,
    IN VOID *SearchKey OPTIONAL,
    IN OUT UINTN *BufferSize,
    OUT EFI_HANDLE *Buffer
    );
```

**Parameters**

- **SearchType** Specifies which handle(s) are to be returned. Type **EFI_LOCATE_SEARCH_TYPE** is defined in “Related Definitions.”
Protocol

Specifies the protocol to search by. This parameter is only valid if SearchType is ByProtocol. Type EFI_GUID is defined in the EFI_BOOT_SERVICES.InstallProtocolInterface() function description.

SearchKey

Specifies the search key. This parameter is ignored if SearchType is AllHandles or ByProtocol. If SearchType is ByRegisterNotify, the parameter must be the Registration value returned by function EFI_BOOT_SERVICES.RegisterProtocolNotify().

BufferSize

On input, the size in bytes of Buffer. On output, the size in bytes of the array returned in Buffer (if the buffer was large enough) or the size, in bytes, of the buffer needed to obtain the array (if the buffer was not large enough).

Buffer

The buffer in which the array is returned. Type EFI_HANDLE is defined in the InstallProtocolInterface() function description.

Related Definitions

```c
typedef enum {  
  AllHandles,  
  ByRegisterNotify,  
  ByProtocol 
} EFI_LOCATE_SEARCH_TYPE;
```

AllHandles

Protocol and SearchKey are ignored and the function returns an array of every handle in the system.

ByRegisterNotify

SearchKey supplies the Registration value returned by EFI_BOOT_SERVICES.RegisterProtocolNotify(). The function returns the next handle that is new for the registration. Only one handle is returned at a time, starting with the first, and the caller must loop until no more handles are returned. Protocol is ignored for this search type.

ByProtocol

All handles that support Protocol are returned. SearchKey is ignored for this search type.

Description

The LocateHandle() function returns an array of handles that match the SearchType request. If the input value of BufferSize is too small, the function returns EFI_BUFFER_TOO_SMALL and updates BufferSize to the size of the buffer needed to obtain the array.
Status Codes Returned

<table>
<thead>
<tr>
<th>Status Code</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>EFI_SUCCESS</td>
<td>The array of handles was returned.</td>
</tr>
<tr>
<td>EFI_NOT_FOUND</td>
<td>No handles match the search.</td>
</tr>
<tr>
<td>EFI_BUFFER_TOO_SMALL</td>
<td>The BufferSize is too small for the result. BufferSize has been updated with the size needed to complete the request.</td>
</tr>
<tr>
<td>EFI_INVALID_PARAMETER</td>
<td>SearchType is not a member of EFI_LOCATE_SEARCH_TYPE.</td>
</tr>
<tr>
<td>EFI_INVALID_PARAMETER</td>
<td>SearchType is ByRegisterNotify and SearchKey is NULL.</td>
</tr>
<tr>
<td>EFI_INVALID_PARAMETER</td>
<td>SearchType is ByProtocol and Protocol is NULL.</td>
</tr>
<tr>
<td>EFI_INVALID_PARAMETER</td>
<td>One or more matches are found and BufferSize is NULL.</td>
</tr>
<tr>
<td>EFI_INVALID_PARAMETER</td>
<td>BufferSize is large enough for the result and Buffer is NULL.</td>
</tr>
</tbody>
</table>

**EFI_BOOT_SERVICES.HandleProtocol()**

Summary
Queries a handle to determine if it supports a specified protocol.

Prototype

```c
typedef EFI_STATUS (EFIAPI *EFI_HANDLE_PROTOCOL) (
    IN EFI_HANDLE Handle,
    IN EFI_GUID *Protocol,
    OUT VOID **Interface
);
```

Parameters

- **Handle**
The handle being queried. If Handle is NULL, then EFI_INVALID_PARAMETER is returned. Type EFI_HANDLE is defined in the EFI_BOOT_SERVICES.InstallProtocolInterface() function description.

- **Protocol**
The published unique identifier of the protocol. It is the caller’s responsibility to pass in a valid GUID. See “Wired For Management Baseline” for a description of valid GUID values. Type EFI_GUID is defined in the InstallProtocolInterface() function description.

- **Interface**
Supplies the address where a pointer to the corresponding Protocol Interface is returned. NULL will be returned in *Interface if a structure is not associated with Protocol.
Description
The `HandleProtocol()` function queries `Handle` to determine if it supports `Protocol`. If it does, then on return `Interface` points to a pointer to the corresponding Protocol Interface. `Interface` can then be passed to any protocol service to identify the context of the request.

EFI 1.10 Extension
The `HandleProtocol()` function is still available for use by old EFI applications and drivers. However, all new applications and drivers should use `EFI_BOOT_SERVICES.OpenProtocol()` in place of `HandleProtocol()`. The following code fragment shows a possible implementation of `HandleProtocol()` using `OpenProtocol()`. The variable `EfiCoreImageHandle` is the image handle of the EFI core.


```c
EFI_STATUS HandleProtocol (  
    IN EFI_HANDLE Handle,  
    IN EFI_GUID *Protocol,  
    OUT VOID **Interface  
)  
{  
    return OpenProtocol (  
        Handle,  
        Protocol,  
        Interface,  
        EfiCoreImageHandle,  
        NULL,  
        EFI_OPEN_PROTOCOL_BY_HANDLE_PROTOCOL  
    );  
}
```

### Status Codes Returned

<table>
<thead>
<tr>
<th>Status Code</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>EFI_SUCCESS</td>
<td>The interface information for the specified protocol was returned.</td>
</tr>
<tr>
<td>EFI_UNSUPPORTED</td>
<td>The device does not support the specified protocol.</td>
</tr>
<tr>
<td>EFI_INVALID_PARAMETER</td>
<td>Handle is <strong>NULL</strong>.</td>
</tr>
<tr>
<td>EFI_INVALID_PARAMETER</td>
<td>Protocol is <strong>NULL</strong>.</td>
</tr>
<tr>
<td>EFI_INVALID_PARAMETER</td>
<td>Interface is <strong>NULL</strong>.</td>
</tr>
</tbody>
</table>

### EFI_BOOT_SERVICES.LocateDevicePath()

#### Summary
Locates the handle to a device on the device path that supports the specified protocol.

#### Prototype

```c
typedef EFI_STATUS  
    (EFIAPI *EFI_LOCATE_DEVICE_PATH) (  
        IN EFI_GUID *Protocol,  
        IN OUT EFI_DEVICE_PATH_PROTOCOL **DevicePath,  
        OUT EFI_HANDLE *Device  
    );
```

#### Parameters

- **Protocol**: The protocol to search for. Type `EFI_GUID` is defined in the `EFI_BOOT_SERVICES.InstallProtocolInterface()` function description.
- **DevicePath**: On input, a pointer to a pointer to the device path. On output, the device path pointer is modified to point to the remaining part of the
device path—that is, when the function finds the closest handle, it splits the device path into two parts, stripping off the front part, and returning the remaining portion. `EFI_DEVICE_PATH_PROTOCOL` is defined in Section 10.2.

**Device**

A pointer to the returned device handle. Type `EFI_HANDLE` is defined in the `InstallProtocolInterface()` function description.

**Description**

The `LocateDevicePath()` function locates all devices on `DevicePath` that support `Protocol` and returns the handle to the device that is closest to `DevicePath`. `DevicePath` is advanced over the device path nodes that were matched.

This function is useful for locating the proper instance of a protocol interface to use from a logical parent device driver. For example, a target device driver may issue the request with its own device path and locate the interfaces to perform I/O on its bus. It can also be used with a device path that contains a file path to strip off the file system portion of the device path, leaving the file path and handle to the file system driver needed to access the file.

If the handle for `DevicePath` supports the protocol (a direct match), the resulting device path is advanced to the device path terminator node. If `DevicePath` is a multi-instance device path, the function will operate on the first instance.

**Status Codes Returned**

<table>
<thead>
<tr>
<th>Code</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>EFI_SUCCESS</td>
<td>The resulting handle was returned.</td>
</tr>
<tr>
<td>EFI_NOT_FOUND</td>
<td>No handles matched the search.</td>
</tr>
<tr>
<td>EFI_INVALID_PARAMETER</td>
<td>Protocol is <code>NULL</code></td>
</tr>
<tr>
<td>EFI_INVALID_PARAMETER</td>
<td><code>DevicePath</code> is <code>NULL</code></td>
</tr>
<tr>
<td>EFI_INVALID_PARAMETER</td>
<td>A handle matched the search and <code>Device</code> is <code>NULL</code>.</td>
</tr>
</tbody>
</table>

** EFI_BOOT_SERVICES.OpenProtocol()**

**Summary**

Queries a handle to determine if it supports a specified protocol. If the protocol is supported by the handle, it opens the protocol on behalf of the calling agent. This is an extended version of the EFI boot service `EFI_BOOT_SERVICES.HandleProtocol()`.
Prototype

```c
typedef EFI_STATUS
  (EFIAPIC *EFI_OPEN_PROTOCOL) ( 
    IN EFI_HANDLE Handle,
    IN EFI_GUID *Protocol,
    OUT VOID **Interface OPTIONAL,
    IN EFI_HANDLE AgentHandle,
    IN EFI_HANDLE ControllerHandle,
    IN UINT32 Attributes)
```

Parameters

- **Handle**
  - The handle for the protocol interface that is being opened.

- **Protocol**
  - The published unique identifier of the protocol. It is the caller’s responsibility to pass in a valid GUID. See “Wired For Management Baseline” for a description of valid GUID values.

- **Interface**
  - Supplies the address where a pointer to the corresponding Protocol Interface is returned. **NULL** will be returned in **Interface** if a structure is not associated with **Protocol**. This parameter is optional, and will be ignored if **Attributes** is **EFI_OPEN_PROTOCOL_TEST_PROTOCOL**.

- **AgentHandle**
  - The handle of the agent that is opening the protocol interface specified by **Protocol** and **Interface**. For agents that follow the UEFI Driver Model, this parameter is the handle that contains the **EFI_DRIVER_BINDING_PROTOCOL** instance that is produced by the UEFI driver that is opening the protocol interface. For UEFI applications, this is the image handle of the UEFI application that is opening the protocol interface. For applications that use **HandleProtocol()** to open a protocol interface, this parameter is the image handle of the EFI firmware.

- **ControllerHandle**
  - If the agent that is opening a protocol is a driver that follows the UEFI Driver Model, then this parameter is the controller handle that requires the protocol interface. If the agent does not follow the UEFI Driver Model, then this parameter is optional and may be **NULL**.

- **Attributes**
  - The open mode of the protocol interface specified by **Handle** and **Protocol**. See "Related Definitions" for the list of legal attributes.

Description

This function opens a protocol interface on the handle specified by **Handle** for the protocol specified by **Protocol**. The first three parameters are the same as **EFI_BOOT_SERVICES.HandleProtocol()**. The only difference is that the agent that is opening a protocol interface is tracked in an EFI's internal handle database. The tracking is used by the UEFI Driver Model, and also used to determine if it is safe to uninstall or reinstall a protocol interface.

The agent that is opening the protocol interface is specified by **AgentHandle**, **ControllerHandle**, and **Attributes**. If the protocol interface can be opened, then **AgentHandle**, **ControllerHandle**, and **Attributes**
are added to the list of agents that are consuming the protocol interface specified by `Handle` and `Protocol`. In addition, the protocol interface is returned in `Interface`, and `EFI_SUCCESS` is returned. If `Attributes` is `TEST_PROTOCOL`, then `Interface` is optional, and can be `NULL`.

There are a number of reasons that this function call can return an error. If an error is returned, then `AgentHandle`, `ControllerHandle`, and `Attributes` are not added to the list of agents consuming the protocol interface specified by `Handle` and `Protocol`. `Interface` is returned unmodified for all error conditions except `EFI_UNSUPPORTED` and `EFI_ALREADY_STARTED`, `NULL` will be returned in `*Interface` when `EFI_UNSUPPORTED` and `Attributes` is not `EFI_OPEN_PROTOCOL_TEST_PROTOCOL`, the protocol interface will be returned in `*Interface` when `EFI_ALREADY_STARTED`.

The following is the list of conditions that must be checked before this function can return `EFI_SUCCESS`:

- If `Protocol` is `NULL`, then `EFI_INVALID_PARAMETER` is returned.
- If `Interface` is `NULL` and `Attributes` is not `TEST_PROTOCOL`, then `EFI_INVALID_PARAMETER` is returned.
- If `Handle` is `NULL`, then `EFI_INVALID_PARAMETER` is returned.
- If `Handle` does not support `Protocol`, then `EFI_UNSUPPORTED` is returned.
- If `Attributes` is not a legal value, then `EFI_INVALID_PARAMETER` is returned. The legal values are listed in “Related Definitions.”
- If `Attributes` is `BY_CHILD_CONTROLLER`, `BY_DRIVER`, `EXCLUSIVE`, or `BY_DRIVER|EXCLUSIVE`, and `AgentHandle` is `NULL`, then `EFI_INVALID_PARAMETER` is returned.
- If `Attributes` is `BY_CHILD_CONTROLLER`, `BY_DRIVER`, `BY_DRIVER|EXCLUSIVE`, and `ControllerHandle` is `NULL`, then `EFI_INVALID_PARAMETER` is returned.
- If `Attributes` is `BY_CHILD_CONTROLLER` and `Handle` is identical to `ControllerHandle`, then `EFI_INVALID_PARAMETER` is returned.
- If `Attributes` is `BY_DRIVER`, `BY_DRIVER|EXCLUSIVE`, or `EXCLUSIVE`, and there are any items on the open list of the protocol interface with an attribute of `EXCLUSIVE` or `BY_DRIVER|EXCLUSIVE`, then `EFI_ACCESS_DENIED` is returned.
- If `Attributes` is `BY_DRIVER`, and there are any items on the open list of the protocol interface with an attribute of `BY_DRIVER`, and `AgentHandle` is the same agent handle in the open list item, then `EFI_ALREADY_STARTED` is returned.
- If `Attributes` is `BY_DRIVER`, and there are any items on the open list of the protocol interface with an attribute of `BY_DRIVER`, and `AgentHandle` is different than the agent handle in the open list item, then `EFI_ACCESS_DENIED` is returned.
- If `Attributes` is `BY_DRIVER|EXCLUSIVE`, and there are any items on the open list of the protocol interface with an attribute of `BY_DRIVER|EXCLUSIVE`, and `AgentHandle` is the same agent handle in the open list item, then `EFI_ALREADY_STARTED` is returned.
- If `Attributes` is `BY_DRIVER|EXCLUSIVE`, and there are any items on the open list of the protocol interface with an attribute of `BY_DRIVER|EXCLUSIVE`, and `AgentHandle` is different than the agent handle in the open list item, then `EFI_ACCESS_DENIED` is returned.
• If `Attributes` is `BY_DRIVER|EXCLUSIVE` or `EXCLUSIVE`, and there is an item on the open list of the protocol interface with an attribute of `BY_DRIVER`, then the boot service `EFI_BOOT_SERVICES.DisconnectController()` is called for the driver on the open list. If there is an item in the open list of the protocol interface with an attribute of `BY_DRIVER` remaining after the `DisconnectController()` call has been made, `EFI_ACCESS_DENIED` is returned.

**Related Definitions**

```c
#define EFI_OPEN_PROTOCOL_BY_HANDLE_PROTOCOL  0x00000001
#define EFI_OPEN_PROTOCOL_GET_PROTOCOL        0x00000002
#define EFI_OPEN_PROTOCOL_TEST_PROTOCOL       0x00000004
#define EFI_OPEN_PROTOCOL_BY_CHILD_CONTROLLER 0x00000008
#define EFI_OPEN_PROTOCOL_BY_DRIVER           0x00000010
#define EFI_OPEN_PROTOCOL_EXCLUSIVE           0x00000020
```

The following is the list of legal values for the `Attributes` parameter, and how each value is used.

- **BY_HANDLE_PROTOCOL** Used in the implementation of `EFI_BOOT_SERVICES.HandleProtocol()`. Since `EFI BOOT SERVICES.OpenProtocol()` performs the same function as `HandleProtocol()` with additional functionality, `HandleProtocol()` can simply call `OpenProtocol()` with this `Attributes` value.

- **GET_PROTOCOL** Used by a driver to get a protocol interface from a handle. Care must be taken when using this open mode because the driver that opens a protocol interface in this manner will not be informed if the protocol interface is uninstalled or reinstalled. The caller is also not required to close the protocol interface with `EFI_BOOT_SERVICES.CloseProtocol()`.

- **TEST_PROTOCOL** Used by a driver to test for the existence of a protocol interface on a handle. `Interface` is optional for this attribute value, so it is ignored, and the caller should only use the return status code. The caller is also not required to close the protocol interface with `CloseProtocol()`.

- **BY_CHILD_CONTROLLER** Used by bus drivers to show that a protocol interface is being used by one of the child controllers of a bus. This information is used by the boot service `EFI_BOOT_SERVICES.ConnectController()` to recursively connect all child controllers and by the boot service `EFI_BOOT_SERVICES.DisconnectController()` to get the list of child controllers that a bus driver created.

- **BY_DRIVER** Used by a driver to gain access to a protocol interface. When this mode is used, the driver's `Stop()` function will be called by `EFI_BOOT_SERVICES.DisconnectController()` if the protocol interface is reinstalled or uninstalled. Once a protocol interface is opened by a driver with this attribute, no other drivers will be allowed to open the same protocol interface with the `BY_DRIVER` attribute.
**BY.Driver|Exclusive** Used by a driver to gain exclusive access to a protocol interface. If any other drivers have the protocol interface opened with an attribute of **BY.Driver**, then an attempt will be made to remove them with `DisconnectController()`.

**Exclusive** Used by applications to gain exclusive access to a protocol interface. If any drivers have the protocol interface opened with an attribute of **BY.Driver**, then an attempt will be made to remove them by calling the driver’s `Stop()` function.
### Status Codes Returned

<table>
<thead>
<tr>
<th>Code</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>EFI_SUCCESS</td>
<td>An item was added to the open list for the protocol interface, and the protocol interface was returned in Interface.</td>
</tr>
<tr>
<td>EFI_INVALID_PARAMETER</td>
<td>Protocol is <strong>NULL</strong>.</td>
</tr>
<tr>
<td>EFI_INVALID_PARAMETER</td>
<td>Interface is <strong>NULL</strong>, and Attributes is not <strong>TEST_PROTOCOL</strong>.</td>
</tr>
<tr>
<td>EFI_INVALID_PARAMETER</td>
<td>Handle is <strong>NULL</strong>.</td>
</tr>
<tr>
<td>EFI_UNSUPPORTED</td>
<td>Handle does not support Protocol.</td>
</tr>
<tr>
<td>EFI_INVALID_PARAMETER</td>
<td>Attributes is not a legal value.</td>
</tr>
<tr>
<td>EFI_INVALID_PARAMETER</td>
<td>Attributes is <strong>BY_CHILD_CONTROLLER</strong> and AgentHandle is <strong>NULL</strong>.</td>
</tr>
<tr>
<td>EFI_INVALID_PARAMETER</td>
<td>Attributes is <strong>BY_DRIVER</strong> and AgentHandle is <strong>NULL</strong>.</td>
</tr>
<tr>
<td>EFI_INVALID_PARAMETER</td>
<td>Attributes is **BY_DRIVER</td>
</tr>
<tr>
<td>EFI_INVALID_PARAMETER</td>
<td>Attributes is <strong>EXCLUSIVE</strong> and AgentHandle is <strong>NULL</strong>.</td>
</tr>
<tr>
<td>EFI_INVALID_PARAMETER</td>
<td>Attributes is <strong>BY_CHILD_CONTROLLER</strong> and ControllerHandle is <strong>NULL</strong>.</td>
</tr>
<tr>
<td>EFI_INVALID_PARAMETER</td>
<td>Attributes is <strong>BY_DRIVER</strong> and ControllerHandle is <strong>NULL</strong>.</td>
</tr>
<tr>
<td>EFI_INVALID_PARAMETER</td>
<td>Attributes is **BY_DRIVER</td>
</tr>
<tr>
<td>EFI_INVALID_PARAMETER</td>
<td>Attributes is <strong>BY_CHILD_CONTROLLER</strong> and Handle is identical to ControllerHandle.</td>
</tr>
<tr>
<td>EFI_ACCESS_DENIED</td>
<td>Attributes is <strong>BY_DRIVER</strong> and there is an item on the open list with an attribute of **BY_DRIVER</td>
</tr>
<tr>
<td>EFI_ACCESS_DENIED</td>
<td>Attributes is **BY_DRIVER</td>
</tr>
<tr>
<td>EFI_ACCESS_DENIED</td>
<td>Attributes is <strong>EXCLUSIVE</strong> and there is an item on the open list with an attribute of **BY_DRIVER</td>
</tr>
<tr>
<td>EFI_ALREADY_STARTED</td>
<td>Attributes is <strong>BY_DRIVER</strong> and there is an item on the open list with an attribute of <strong>BY_DRIVER</strong> whose agent handle is the same as AgentHandle.</td>
</tr>
<tr>
<td>EFI_ACCESS_DENIED</td>
<td>Attributes is <strong>BY_DRIVER</strong> and there is an item on the open list with an attribute of <strong>BY_DRIVER</strong> whose agent handle is different than AgentHandle.</td>
</tr>
<tr>
<td>EFI_ALREADY_STARTED</td>
<td>Attributes is **BY_DRIVER</td>
</tr>
<tr>
<td>EFI_ACCESS_DENIED</td>
<td>Attributes is **BY_DRIVER</td>
</tr>
<tr>
<td>EFI_ACCESS_DENIED</td>
<td>Attributes is **BY_DRIVER</td>
</tr>
</tbody>
</table>
Examples

```c
EFI_BOOT_SERVICES    *gBS;
EFI_HANDLE           ImageHandle;
EFI_DRIVER_BINDING_PROTOCOL   *This;
IN EFI_HANDLE     ControllerHandle,
extern EFI_GUID   gEfiXyzIoProtocol;
EFI_XYZ_IO_PROTOCOL  *XyzIo;
EFI_STATUS          Status;
```

/// EFI_OPEN_PROTOCOL_BY_HANDLE_PROTOCOL example
/// Retrieves the XYZ I/O Protocol instance from ControllerHandle
/// The application that is opening the protocol is identified by ImageHandle
/// Possible return status codes:
/// EFI_SUCCESS : The protocol was opened and returned in XyzIo
/// EFI_UNSUPPORTED : The protocol is not present on ControllerHandle
/// Status = gBS->OpenProtocol (
// ControllerHandle,
// &gEfiXyzIoProtocol,
// &XyzIo,
// ImageHandle,
// NULL,
// EFI_OPEN_PROTOCOL_BY_HANDLE_PROTOCOL);

/// EFI_OPEN_PROTOCOL_GET_PROTOCOL example
/// Retrieves the XYZ I/O Protocol instance from ControllerHandle
/// The driver that is opening the protocol is identified by the
/// Driver Binding Protocol instance This. This->DriverBindingHandle
/// identifies the agent that is opening the protocol interface, and it
/// is opening this protocol on behalf of ControllerHandle.
/// Possible return status codes:
/// EFI_SUCCESS : The protocol was opened and returned in XyzIo
/// EFI_UNSUPPORTED : The protocol is not present on ControllerHandle
/// Status = gBS->OpenProtocol (
// ControllerHandle,
// &gEfiXyzIoProtocol,
// &XyzIo,
// This->DriverBindingHandle,
// ControllerHandle,
// EFI_OPEN_PROTOCOL_GET_PROTOCOL);

/// EFI_OPEN_PROTOCOL_TEST_PROTOCOL example
/// Tests to see if the XYZ I/O Protocol is present on ControllerHandle
/// The driver that is opening the protocol is identified by the
/// Driver Binding Protocol instance This. This->DriverBindingHandle
/// identifies the agent that is opening the protocol interface, and it
/// is opening this protocol on behalf of ControllerHandle.
/// EFI_SUCCESS : The protocol was opened and returned in XyzIo
/// EFI_UNSUPPORTED : The protocol is not present on ControllerHandle
// Status = gBS->OpenProtocol (  
ControllerHandle,  
&gEfiXyzIoProtocol,  
NULL,  
This->DriverBindingHandle,  
ControllerHandle,  
EFI_OPEN_PROTOCOL_TEST_PROTOCOL  
);

// EFI_OPEN_PROTOCOL_BY_DRIVER example  
// Opens the XYZ I/O Protocol on ControllerHandle  
// The driver that is opening the protocol is identified by the  
// Driver Binding Protocol instance This. This->DriverBindingHandle  
// identifies the agent that is opening the protocol interface, and it  
// is opening this protocol on behalf of ControllerHandle.  
// Possible return status codes:  
// EFI_SUCCESS : The protocol was opened and returned in XyzIo  
// EFI_UNSUPPORTED : The protocol is not present on ControllerHandle  
// EFI_ALREADY_STARTED : The protocol is already opened by the driver  
// EFI_ACCESS_DENIED : The protocol is managed by a different driver  
// Status = gBS->OpenProtocol (  
ControllerHandle,  
&gEfiXyzIoProtocol,  
&XyzIo,  
This->DriverBindingHandle,  
ControllerHandle,  
EFI_OPEN_PROTOCOL_BY_DRIVER  
);

// EFI_OPEN_PROTOCOL_BY_DRIVER | EFI_OPEN_PROTOCOL_EXCLUSIVE example  
// Opens the XYZ I/O Protocol on ControllerHandle  
// The driver that is opening the protocol is identified by the  
// Driver Binding Protocol instance This. This->DriverBindingHandle  
// identifies the agent that is opening the protocol interface, and it  
// is opening this protocol on behalf of ControllerHandle.  
// Possible return status codes:  
// EFI_SUCCESS : The protocol was opened and returned in XyzIo. If //  
// a different driver had the XYZ I/O Protocol opened  
// BY_DRIVER, then that driver was disconnected to  
// allow this driver to open the XYZ I/O Protocol.  
// EFI_UNSUPPORTED : The protocol is not present on ControllerHandle  
// EFI_ALREADY_STARTED : The protocol is already opened by the driver  
// EFI_ACCESS_DENIED : The protocol is managed by a different driver that //  
// already has the protocol opened with an EXCLUSIVE // attribute.  
// Status = gBS->OpenProtocol (  
ControllerHandle,  
&gEfiXyzIoProtocol,  
&XyzIo,  
This->DriverBindingHandle,  
ControllerHandle,


```c
EFI_OPEN_PROTOCOL_BY_DRIVER | EFI_OPEN_PROTOCOL_EXCLUSIVE
```

**EFI_BOOT_SERVICES.CloseProtocol()**

**Summary**
Closes a protocol on a handle that was opened using `EFI_BOOT_SERVICES.OpenProtocol()`.

**Prototype**
```c
typedef
EFI_STATUS
(EIFIAPI *EFI_CLOSE_PROTOCOL) (  
    IN EFI_HANDLE Handle,  
    IN EFI_GUID *Protocol,  
    IN EFI_HANDLE AgentHandle,  
    IN EFI_HANDLE ControllerHandle
);
```

**Parameters**
- **Handle**
  The handle for the protocol interface that was previously opened with `OpenProtocol()`, and is now being closed.

- **Protocol**
  The published unique identifier of the protocol. It is the caller’s responsibility to pass in a valid GUID. See “Wired For Management Baseline” for a description of valid GUID values.

- **AgentHandle**
  The handle of the agent that is closing the protocol interface. For agents that follow the UEFI Driver Model, this parameter is the handle that contains the `EFI_DRIVER_BINDING_PROTOCOL` instance that is produced by the UEFI driver that is opening the protocol interface. For UEFI applications, this is the image handle of the UEFI application. For applications that used `EFI_BOOT_SERVICES.HandleProtocol()` to open the protocol interface, this will be the image handle of the EFI firmware.

- **ControllerHandle**
  If the agent that opened a protocol is a driver that follows the UEFI Driver Model, then this parameter is the controller handle that required the protocol interface. If the agent does not follow the UEFI Driver Model, then this parameter is optional and may be `NULL`.

**Description**
This function updates the handle database to show that the protocol instance specified by `Handle` and `Protocol` is no longer required by the agent and controller specified `AgentHandle` and `ControllerHandle`.

If `Handle` or `AgentHandle` is `NULL`, then `EFI_INVALID_PARAMETER` is returned. If `ControllerHandle` is not `NULL`, and `ControllerHandle` is `NULL`, then `EFI_INVALID_PARAMETER` is returned. If `Protocol` is `NULL`, then `EFI_INVALID_PARAMETER` is returned.

If the interface specified by `Protocol` is not supported by the handle specified by `Handle`, then `EFI_NOT_FOUND` is returned.
If the interface specified by `Protocol` is supported by the handle specified by `Handle`, then a check is made to see if the protocol instance specified by `Protocol` and `Handle` was opened by `AgentHandle` and `ControllerHandle` with `EFI_BOOT_SERVICES.OpenProtocol()`. If the protocol instance was not opened by `AgentHandle` and `ControllerHandle`, then `EFI_NOT_FOUND` is returned. If the protocol instance was opened by `AgentHandle` and `ControllerHandle`, then all of those references are removed from the handle database, and `EFI_SUCCESS` is returned.

### Status Codes Returned

<table>
<thead>
<tr>
<th>Code</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>EFI_SUCCESS</td>
<td>The protocol instance was closed.</td>
</tr>
<tr>
<td>EFI_INVALID_PARAMETER</td>
<td><code>Handle</code> is <code>NULL</code>.</td>
</tr>
<tr>
<td>EFI_INVALID_PARAMETER</td>
<td><code>AgentHandle</code> is <code>NULL</code>.</td>
</tr>
<tr>
<td>EFI_INVALID_PARAMETER</td>
<td><code>ControllerHandle</code> is not <code>NULL</code> and <code>ControllerHandle</code> is <code>NULL</code>.</td>
</tr>
<tr>
<td>EFI_INVALID_PARAMETER</td>
<td><code>Protocol</code> is <code>NULL</code>.</td>
</tr>
<tr>
<td>EFI_NOT_FOUND</td>
<td><code>Handle</code> does not support the protocol specified by <code>Protocol</code>.</td>
</tr>
<tr>
<td>EFI_NOT_FOUND</td>
<td>The protocol interface specified by <code>Handle</code> and <code>Protocol</code> is not currently open by <code>AgentHandle</code> and <code>ControllerHandle</code>.</td>
</tr>
</tbody>
</table>

### Examples

```c
#include <stdio.h>

int main() {
    // EFI_BOOT_SERVICES *gBS;
    // EFI_HANDLE ImageHandle;
    // EFI_DRIVER_BINDING_PROTOCOL *This;
    // IN EFI_HANDLE ControllerHandle,
    // extern EFI_GUID gEfiXyzIoProtocol;
    // EFI_STATUS Status;

    // Close the XYZ I/O Protocol that was opened on behalf of ControllerHandle
    Status = gBS->CloseProtocol(
        ControllerHandle,
        &gEfiXyzIoProtocol,
        This->DriverBindingHandle,
        ControllerHandle
    );
    return Status;
}
```


UEFI Specification, Version 2.9
Services — Boot Services

// Close the XYZ I/O Protocol that was opened with BY_HANDLE_PROTOCOL
// Status = gBS->CloseProtocol (ControllerHandle, &gEfiXyzIoProtocol, ImageHandle, NULL);

EFI_BOOT_SERVICES.OpenProtocolInformation()

Summary
Retrieves the list of agents that currently have a protocol interface opened.

Prototype
typedef EFI_STATUS
(EFIAPI *EFI_OPEN_PROTOCOL_INFORMATION) (  
    IN EFI_HANDLE Handle,  
    IN EFI_GUID *Protocol,  
    OUT EFI_OPEN_PROTOCOL_INFORMATION_ENTRY **EntryBuffer,  
    OUT UINTN *EntryCount
);

Parameters

Handle  The handle for the protocol interface that is being queried.
Protocol  The published unique identifier of the protocol. It is the caller's responsibility to pass in a valid GUID. See “Wired For Management Baseline” for a description of valid GUID values.
EntryBuffer  A pointer to a buffer of open protocol information in the form of EFI_OPEN_PROTOCOL_INFORMATION_ENTRY structures. See "Related Definitions" for the declaration of this type. The buffer is allocated by this service, and it is the caller's responsibility to free this buffer when the caller no longer requires the buffer's contents.
EntryCount  A pointer to the number of entries in EntryBuffer.
Related Definitions

typedef struct {
  EFI_HANDLE AgentHandle;
  EFI_HANDLE ControllerHandle;
  UINT32 Attributes;
  UINT32 OpenCount;
} EFI_OPEN_PROTOCOL_INFORMATION_ENTRY;

Description

This function allocates and returns a buffer of EFI_OPEN_PROTOCOL_INFORMATION_ENTRY structures. The buffer is returned in EntryBuffer, and the number of entries is returned in EntryCount.

If the interface specified by Protocol is not supported by the handle specified by Handle, then EFI_NOT_FOUND is returned.

If the interface specified by Protocol is supported by the handle specified by Handle, then EntryBuffer is allocated with the boot service EFI_BOOT_SERVICES.AllocatePool(), and EntryCount is set to the number of entries in EntryBuffer. Each entry of EntryBuffer is filled in with the image handle, controller handle, and attributes that were passed to EFI_BOOT_SERVICES.OpenProtocol() when the protocol interface was opened. The field OpenCount shows the number of times that the protocol interface has been opened by the agent specified by ImageHandle, ControllerHandle, and Attributes. After the contents of EntryBuffer have been filled in, EFI_SUCCESS is returned. It is the caller’s responsibility to call EFI_BOOT_SERVICES.FreePool() on EntryBuffer when the caller no longer required the contents of EntryBuffer.

If there are not enough resources available to allocate EntryBuffer, then EFI_OUT_OF_RESOURCES is returned.

Status Codes Returned

<table>
<thead>
<tr>
<th>Status Code</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>EFI_SUCCESS</td>
<td>The open protocol information was returned in EntryBuffer, and the number of entries was returned EntryCount.</td>
</tr>
<tr>
<td>EFI_NOT_FOUND</td>
<td>Handle does not support the protocol specified by Protocol.</td>
</tr>
<tr>
<td>EFI_OUT_OF_RESOURCES</td>
<td>There are not enough resources available to allocate EntryBuffer.</td>
</tr>
</tbody>
</table>

Examples

See example in the EFI_BOOT_SERVICES.LocateHandleBuffer() function description for an example on how LocateHandleBuffer(), EFI_BOOT_SERVICES.ProtocolsPerHandle(), OpenProtocol(), and EFI_BOOT_SERVICES.OpenProtocolInformation() can be used to traverse the entire handle database.

EFI_BOOT_SERVICES.ConnectController()

Summary

Connects one or more drivers to a controller.

Prototype

typedef
EFI_STATUS
(EIFI_API *EFI_CONNECT_CONTROLLER) (
  IN EFI_HANDLE ControllerHandle,
  IN EFI_HANDLE *DriverImageHandle OPTIONAL,
  IN EFI_DEVICE_PATH_PROTOCOL *RemainingDevicePath OPTIONAL,
  IN BOOLEAN Recursive
);

Parameters

ControllerHandle The handle of the controller to which driver(s) are to be connected.

DriverImageHandle A pointer to an ordered list handles that support the EFI_DRIVER_BINDING_PROTOCOL. The list is terminated by a NULL handle value. These handles are candidates for the Driver Binding Protocol(s) that will manage the controller specified by ControllerHandle. This is an optional parameter that may be NULL. This parameter is typically used to debug new drivers.

RemainingDevicePath A pointer to the device path that specifies a child of the controller specified by ControllerHandle. This is an optional parameter that may be NULL. If it is NULL, then handles for all the children of ControllerHandle will be created. This parameter is passed unchanged to the Supported() and Start() services of the EFI_DRIVER_BINDING_PROTOCOL attached to ControllerHandle.

Recursive If TRUE, then ConnectController() is called recursively until the entire tree of controllers below the controller specified by ControllerHandle have been created. If FALSE, then the tree of controllers is only expanded one level.

Description

This function connects one or more drivers to the controller specified by ControllerHandle. If ControllerHandle is NULL, then EFI_INVALID_PARAMETER is returned. If there are no EFI_DRIVER_BINDING_PROTOCOL instances present in the system, then return EFI_NOT_FOUND. If there are not enough resources available to complete this function, then EFI_OUT_OF_RESOURCES is returned.

If the platform supports user authentication, as specified in Section 36, the device path associated with ControllerHandle is checked against the connect permissions in the current user profile. If forbidden, then EFI_SECURITY_VIOLATION is returned. Then, before connecting any of the DriverImageHandles, the device path associated with the handle is checked against the connect permissions in the current user profile.

If Recursive is FALSE, then this function returns after all drivers have been connected to ControllerHandle. If Recursive is TRUE, then ConnectController() is called recursively on all of the child controllers of ControllerHandle. The child controllers can be identified by searching the handle database for all the controllers that have opened ControllerHandle with an attribute of EFI_OPEN_PROTOCOL_BY_CHILD_CONTROLLER.

This function uses five precedence rules when deciding the order that drivers are tested against controllers. These five rules from highest precedence to lowest precedence are as follows:
1. **Context Override**: `DriverImageHandle` is an ordered list of handles that support the `EFI_DRIVER_BINDING_PROTOCOL`. The highest priority image handle is the first element of the list, and the lowest priority image handle is the last element of the list. The list is terminated with a `NULL` image handle.

2. **Platform Driver Override**: If an `EFI_PLATFORM_DRIVER_OVERRIDE_PROTOCOL` instance is present in the system, then the `GetDriver()` service of this protocol is used to retrieve an ordered list of image handles for `ControllerHandle`. From this list, the image handles found in rule (1) above are removed. The first image handle returned from `GetDriver()` has the highest precedence, and the last image handle returned from `GetDriver()` has the lowest precedence. The ordered list is terminated when `GetDriver()` returns `EFI_NOT_FOUND`. It is legal for no image handles to be returned by `GetDriver()`. There can be at most a single instance in the system of the `EFI_PLATFORM_DRIVER_OVERRIDE_PROTOCOL`. If there is more than one, then the system behavior is not deterministic.

3. **Driver Family Override Search**: The list of available driver image handles can be found by using the boot service `EFI_BOOT_SERVICES.LocateHandle()` with a `SearchType` of `ByProtocol` for the GUID of the `EFI_DRIVER_FAMILY_OVERRIDE_PROTOCOL`. From this list, the image handles found in rules (1), and (2) above are removed. The remaining image handles are sorted from highest to lowest based on the value returned from the `GetVersion()` function of the `EFI_DRIVER_FAMILY_OVERRIDE_PROTOCOL` associated with each image handle.

4. **Bus Specific Driver Override**: If there is an instance of the `EFI_BUS_SPECIFIC_DRIVER_OVERRIDE_PROTOCOL` attached to `ControllerHandle`, then the `GetDriver()` service of this protocol is used to retrieve an ordered list of image handle for `ControllerHandle`. From this list, the image handles found in rules (1), (2), and (3) above are removed. The first image handle returned from `GetDriver()` has the highest precedence, and the last image handle returned from `GetDriver()` has the lowest precedence. The ordered list is terminated when `GetDriver()` returns `EFI_NOT_FOUND`. It is legal for no image handles to be returned by `GetDriver()`.

5. **Driver Binding Search**: The list of available driver image handles can be found by using the boot service `EFI_BOOT_SERVICES.LocateHandle()` with a `SearchType` of `ByProtocol` for the GUID of the `EFI_DRIVER_BINDING_PROTOCOL`. From this list, the image handles found in rules (1), (2), (3), and (4) above are removed. The remaining image handles are sorted from highest to lowest based on the `Version` field of the `EFI_DRIVER_BINDING_PROTOCOL` instance associated with each image handle.

Each of the five groups of image handles listed above is tested against `ControllerHandle` in order by using the `EFI_DRIVER_BINDING_PROTOCOL` service `Supported()`. `RemainingDevicePath` is passed into `Supported()` unmodified. The first image handle whose `Supported()` service returns `EFI_SUCCESS` is marked so the image handle will not be tried again during this call to `ConnectController()`. Then, the `Start()` service of the `EFI_DRIVER_BINDING_PROTOCOL` is called for `ControllerHandle`. Once again, `RemainingDevicePath` is passed in unmodified. Every time `Supported()` returns `EFI_SUCCESS`, the search for drivers restarts with the highest precedence image handle. This process is repeated until no image handles pass the `Supported()` check.

If at least one image handle returned `EFI_SUCCESS` from its `Start()` service, then `EFI_SUCCESS` is returned.

If no image handles returned `EFI_SUCCESS` from their `Start()` service then `EFI_NOT_FOUND` is returned unless `RemainingDevicePath` is not `NULL`, and `RemainingDevicePath` is an End Node. In this
special case, **EFI_SUCCESS** is returned because it is not an error to fail to start a child controller that is specified by an End Device Path Node.

### Status Codes Returned

<table>
<thead>
<tr>
<th>Status Code</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>EFI_SUCCESS</td>
<td>One or more drivers were connected to <code>ControllerHandle</code>.</td>
</tr>
<tr>
<td>EFI_SUCCESS</td>
<td>No drivers were connected to <code>ControllerHandle</code>, but <code>RemainingDevicePath</code> is not <code>NULL</code>, and it is an End Device Path Node.</td>
</tr>
<tr>
<td>EFI_INVALID_PARAMETER</td>
<td><code>ControllerHandle</code> is <code>NULL</code>.</td>
</tr>
<tr>
<td>EFI_NOT_FOUND</td>
<td>There are no <code>EFI_DRIVER_BINDING_PROTOCOL</code> instances present in the system.</td>
</tr>
<tr>
<td>EFI_NOT_FOUND</td>
<td>No drivers were connected to <code>ControllerHandle</code>.</td>
</tr>
<tr>
<td>EFI_SECURITY_VIOLATION</td>
<td>The user has no permission to start UEFI device drivers on the device path associated with the <code>ControllerHandle</code> or specified by the <code>RemainingDevicePath</code>.</td>
</tr>
</tbody>
</table>

### Examples

```c
//
// Connect All Handles Example
// The following example recursively connects all controllers in a platform.
//
EFI_STATUS Status;
EFI_BOOT_SERVICES *gBS;
UINTN HandleCount;
EFI_HANDLE *HandleBuffer;
UINTN HandleIndex;

// Retrieve the list of all handles from the handle database
// Status = gBS->LocateHandleBuffer (AllHandles, NULL, NULL, &HandleCount, &HandleBuffer);
if (!EFI_ERROR (Status)) {
    for (HandleIndex = 0; HandleIndex < HandleCount; HandleIndex++) {
        Status = gBS->ConnectController (HandleBuffer[HandleIndex], NULL, NULL, TRUE);
    }
    gBS->FreePool(HandleBuffer);
}
//
```
// Connect Device Path Example
// The following example walks the device path nodes of a device path, and
// connects only the drivers required to force a handle with that device path
// to be present in the handle database. This algorithms guarantees that
// only the minimum number of devices and drivers are initialized.
//
EFI_STATUS Status;
EFI_DEVICE_PATH_PROTOCOL *DevicePath;
EFI_DEVICE_PATH_PROTOCOL *RemainingDevicePath;
EFI_HANDLE Handle;

do {
    // Find the handle that best matches the Device Path. If it is only a
    // partial match the remaining part of the device path is returned in
    // RemainingDevicePath.
    RemainingDevicePath = DevicePath;
    Status = gBS->LocateDevicePath (
        &gEfiDevicePathProtocolGuid,
        &RemainingDevicePath,
        &Handle
    );
    if (EFI_ERROR(Status)) {
        return EFI_NOT_FOUND;
    }

    // Connect all drivers that apply to Handle and RemainingDevicePath
    // If no drivers are connected Handle, then return EFI_NOT_FOUND
    // The Recursive flag is FALSE so only one level will be expanded.
    Status = gBS->ConnectController (
        Handle,
        NULL,
        RemainingDevicePath,
        FALSE
    );
    if (EFI_ERROR(Status)) {
        return EFI_NOT_FOUND;
    }

    // Loop until RemainingDevicePath is an empty device path
    //
} while (!IsDevicePathEnd (RemainingDevicePath));

// A handle with DevicePath exists in the handle database
// return EFI_SUCCESS;
**EFI_BOOT_SERVICES.DisconnectController()**

**Summary**
Disconnects one or more drivers from a controller.

**Prototype**

```c
typedef EFI_STATUS
  (EFIAPI *EFI_DISCONNECT_CONTROLLER) (  
    IN EFI_HANDLE ControllerHandle,  
    IN EFI_HANDLE DriverImageHandle OPTIONAL,  
    IN EFI_HANDLE ChildHandle OPTIONAL  
  );
```

**Parameters**

- **ControllerHandle** The handle of the controller from which driver(s) are to be disconnected.
- **DriverImageHandle** The driver to disconnect from ControllerHandle. If DriverImageHandle is NULL, then all the drivers currently managing ControllerHandle are disconnected from ControllerHandle. If DriverImageHandle is not NULL, and DriverImageHandle is not a valid EFI_HANDLE, then EFI_INVALID_PARAMETER is returned. If DriverImageHandle is not NULL, and DriverImageHandle is not currently managing ControllerHandle, then EFI_SUCCESS is returned. If ChildHandle is not NULL, and DriverImageHandle is a valid EFI_HANDLE, then EFI_INVALID_PARAMETER is returned. If there are not enough resources available to disconnect drivers from ControllerHandle, then EFI_OUT_OF_RESOURCES is returned.
- **ChildHandle** The handle of the child to destroy. If ChildHandle is NULL, then all the children of ControllerHandle are destroyed before the drivers are disconnected from ControllerHandle.

**Description**
This function disconnects one or more drivers from the controller specified by ControllerHandle. If DriverImageHandle is NULL, then all of the drivers currently managing ControllerHandle are disconnected from ControllerHandle. If DriverImageHandle is not NULL, then only the driver specified by DriverImageHandle is disconnected from ControllerHandle. If ChildHandle is NULL, then all of the children of ControllerHandle are destroyed before the drivers are disconnected from ControllerHandle. If ChildHandle is not NULL, then only the child controller specified by ChildHandle is destroyed. If ChildHandle is the only child of ControllerHandle, then the driver specified by DriverImageHandle will be disconnected from ControllerHandle. A driver is disconnected from a controller by calling the **Stop()** service of the **EFI_DRIVER_BINDING_PROTOCOL**. The **EFI_DRIVER_BINDING_PROTOCOL** is on the driver image handle, and the handle of the controller is passed into the **Stop()** service. The list of drivers managing a controller, and the list of children for a specific controller can be retrieved from the handle database with the boot service **EFI_BOOT_SERVICES.OpenProtocolInformation()**. If all the required drivers are disconnected from ControllerHandle, then EFI_SUCCESS is returned.

If ControllerHandle is NULL, then EFI_INVALID_PARAMETER is returned. If no drivers are managing ControllerHandle, then EFI_SUCCESS is returned. If DriverImageHandle is not NULL, and DriverImageHandle is not a valid EFI_HANDLE, then EFI_INVALID_PARAMETER is returned. If DriverImageHandle is not NULL, and DriverImageHandle is not currently managing ControllerHandle, then EFI_SUCCESS is returned. If ChildHandle is not NULL, and ChildHandle is not a valid EFI_HANDLE, then EFI_INVALID_PARAMETER is returned. If there are not enough resources available to disconnect drivers from ControllerHandle, then EFI_OUT_OF_RESOURCES is returned.
Status Codes Returned

<table>
<thead>
<tr>
<th>Code</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>EFI_SUCCESS</td>
<td>One or more drivers were disconnected from the controller.</td>
</tr>
<tr>
<td>EFI_SUCCESS</td>
<td>On entry, no drivers are managing ControllerHandle.</td>
</tr>
<tr>
<td>EFI_SUCCESS</td>
<td>DriverImageHandle is not NULL, and on entry DriverImageHandle is not managing ControllerHandle.</td>
</tr>
<tr>
<td>EFI_INVALID_PARAMETER</td>
<td>ControllerHandle is NULL.</td>
</tr>
<tr>
<td>EFI_INVALID_PARAMETER</td>
<td>DriverImageHandle is not NULL, and it is not a valid EFI_HANDLE.</td>
</tr>
<tr>
<td>EFI_INVALID_PARAMETER</td>
<td>ChildHandle is not NULL, and it is not a valid EFI_HANDLE.</td>
</tr>
<tr>
<td>EFI_OUT_OF_RESOURCES</td>
<td>There are not enough resources available to disconnect any drivers from ControllerHandle.</td>
</tr>
<tr>
<td>EFI_DEVICE_ERROR</td>
<td>The controller could not be disconnected because of a device error.</td>
</tr>
<tr>
<td>EFI_INVALID_PARAMETER</td>
<td>DriverImageHandle does not support the EFI_DRIVER_BINDING_PROTOCOL.</td>
</tr>
</tbody>
</table>

Examples

```c
//
// Disconnect All Handles Example
// The following example recursively disconnects all drivers from all controllers in a platform.
//

EFI_STATUS    Status;
EFI_BOOT_SERVICES  *gBS;
UINTN                HandleCount;
EFI_HANDLE       *HandleBuffer;
UINTN                HandleIndex;

// Retrieve the list of all handles from the handle database
//
Status = gBS->LocateHandleBuffer (   
    AllHandles,   
    NULL,       
    NULL,       
    &HandleCount,   
    &HandleBuffer
);
if (!EFI_ERROR (Status)) {
    for (HandleIndex = 0; HandleIndex < HandleCount; HandleIndex++) {
        Status = gBS->DisconnectController (   
            HandleBuffer[HandleIndex],   
            NULL,                       
            NULL
        );
    }
}
gBS->FreePool(HandleBuffer);
```
**EFI_BOOT_SERVICES.ProtocolsPerHandle()**

**Summary**
Retrieves the list of protocol interface GUIDs that are installed on a handle in a buffer allocated from pool.

**Prototype**
```c
typedef EFI_STATUS
  (EFIAPI *EFI_PROTOCOLS_PER_HANDLE) (
    IN EFI_HANDLE Handle,
    OUT EFI_GUID ***ProtocolBuffer,
    OUT UINTN   *ProtocolBufferCount
  );
```

**Parameters**
- **Handle**  The handle from which to retrieve the list of protocol interface GUIDs.
- **ProtocolBuffer**  A pointer to the list of protocol interface GUID pointers that are installed on `Handle`. This buffer is allocated with a call to the Boot Service `EFI_BOOT_SERVICES.AllocatePool()`. It is the caller's responsibility to call the Boot Service `EFI_BOOT_SERVICES.FreePool()` when the caller no longer requires the contents of `ProtocolBuffer`.
- **ProtocolBufferCount**  A pointer to the number of GUID pointers present in `ProtocolBuffer`.

**Description**
The `ProtocolsPerHandle()` function retrieves the list of protocol interface GUIDs that are installed on `Handle`. The list is returned in `ProtocolBuffer`, and the number of GUID pointers in `ProtocolBuffer` is returned in `ProtocolBufferCount`.

If `Handle` is **NULL** or `Handle` is **NULL**, then `EFI_INVALID_PARAMETER` is returned.

If `ProtocolBuffer` is **NULL**, then `EFI_INVALID_PARAMETER` is returned.

If `ProtocolBufferCount` is **NULL**, then `EFI_INVALID_PARAMETER` is returned.

If there are not enough resources available to allocate `ProtocolBuffer`, then `EFI_OUT_OF_RESOURCES` is returned.
Status Codes Returned

<table>
<thead>
<tr>
<th>Code</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>EFI_SUCCESS</td>
<td>The list of protocol interface GUIDs installed on Handle was returned in</td>
</tr>
<tr>
<td></td>
<td>ProtocolBuffer. The number of protocol interface GUIDs was returned in</td>
</tr>
<tr>
<td></td>
<td>ProtocolBufferCount.</td>
</tr>
<tr>
<td>EFI_INVALID_PARAMETER</td>
<td>Handle is NULL.</td>
</tr>
<tr>
<td>EFI_INVALID_PARAMETER</td>
<td>ProtocolBuffer is NULL.</td>
</tr>
<tr>
<td>EFI_INVALID_PARAMETER</td>
<td>ProtocolBufferCount is NULL.</td>
</tr>
<tr>
<td>EFI_OUT_OF_RESOURCES</td>
<td>There is not enough pool memory to store the results.</td>
</tr>
</tbody>
</table>

Examples

See example in the EFI_BOOT_SERVICES.LocateHandleBuffer() function description for an example on how LocateHandleBuffer(), EFI_BOOT_SERVICES.ProtocolsPerHandle(), EFI_BOOT_SERVICES.OpenProtocol(), and EFI_BOOT_SERVICES.OpenProtocolInformation() can be used to traverse the entire handle database.

EFI_BOOT_SERVICES.LocateHandleBuffer()

Summary

Returns an array of handles that support the requested protocol in a buffer allocated from pool.

Prototype

```c
typedef
EFI_STATUS
(EIFI_API *EFI_LOCATE_HANDLE_BUFFER) (    
    IN EFI_LOCATE_SEARCH_TYPE SearchType,    
    IN EFI_GUID *Protocol OPTIONAL,        
    IN VOID *SearchKey OPTIONAL,            
    OUT UINTN *NoHandles,                   
    OUT EFI_HANDLE **Buffer                
);
```

Parameters

- **SearchType** Specifies which handle(s) are to be returned.
- **Protocol** Provides the protocol to search by. This parameter is only valid for a SearchType of ByProtocol.
- **SearchKey** Supplies the search key depending on the SearchType.
- **NoHandles** The number of handles returned in Buffer.
- **Buffer** A pointer to the buffer to return the requested array of handles that support Protocol. This buffer is allocated with a call to the Boot Service EFI_BOOT_SERVICES.AllocatePool(). It is the caller's responsibility to call the Boot Service EFI_BOOT_SERVICES.FreePool() when the caller no longer requires the contents of Buffer.
Description

The LocateHandleBuffer() function returns one or more handles that match the SearchType request. Buffer is allocated from pool, and the number of entries in Buffer is returned in NoHandles. Each SearchType is described below:

- **AllHandles**
  - Protocol and SearchKey are ignored and the function returns an array of every handle in the system.

- **ByRegisterNotify**
  - SearchKey supplies the Registration returned by EFI_BOOT_SERVICES.RegisterProtocolNotify(). The function returns the next handle that is new for the Registration. Only one handle is returned at a time, and the caller must loop until no more handles are returned. Protocol is ignored for this search type.

- **ByProtocol**
  - All handles that support Protocol are returned. SearchKey is ignored for this search type.

If NoHandles is NULL, then EFI_INVALID_PARAMETER is returned.

If Buffer is NULL, then EFI_INVALID_PARAMETER is returned.

If there are no handles in the handle database that match the search criteria, then EFI_NOT_FOUND is returned.

If there are not enough resources available to allocate Buffer, then EFI_OUT_OF_RESOURCES is returned.

Status Codes Returned

<table>
<thead>
<tr>
<th>Status Code</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>EFI_SUCCESS</td>
<td>The array of handles was returned in Buffer, and the number of handles in Buffer was returned in NoHandles.</td>
</tr>
<tr>
<td>EFI_INVALID_PARAMETER</td>
<td>NoHandles is NULL</td>
</tr>
<tr>
<td>EFI_INVALID_PARAMETER</td>
<td>Buffer is NULL</td>
</tr>
<tr>
<td>EFI_NOT_FOUND</td>
<td>No handles match the search.</td>
</tr>
<tr>
<td>EFI_OUT_OF_RESOURCES</td>
<td>There is not enough pool memory to store the matching results.</td>
</tr>
</tbody>
</table>

Examples

```c
EFI_STATUS Status;
EFI_BOOT_SERVICES *gBS;
EFI_HANDLE ImageHandle;
```
// Retrieve the list of all handles from the handle database
//
status = gbs->locatehandlebuffer (HANDLE, NULL, NULL, &handlecount, &handlebuffer);
if (!EFI_ERROR (status)) {
    for (handleindex = 0; handleindex < handlecount; handleindex++) {
        // Retrieve the list of all the protocols on each handle
        //
        status = gbs->protocolsperhandle (HANDLE, &protocolguidarray, &arraycount);
        if (!EFI_ERROR (status)) {
            for (protocolindex = 0; protocolindex < arraycount; protocolindex++) {
                // Retrieve the protocol instance for each protocol
                //
                status = gbs->openprotocol (HANDLE, protocolguidarray[protocolindex], &instance, NULL, NULL, EFI_OPEN_PROTOCOL_GET_PROTOCOL);
            }
        }
    }
}
// Retrieve the list of agents that have opened each protocol
//
status = gbs->openprotocolinformation (HANDLE, protocolguidarray[protocolindex], &openinfo, &openinfocount);
if (!EFI_ERROR (status)) {
    for (openinfoindex = 0; openinfoindex < openinfocount; openinfoindex++) {

EFI_BOOT_SERVICES.LocateProtocol()

Summary
Returns the first protocol instance that matches the given protocol.

Prototype

typedef
EFI_STATUS
(EIFIAPIC *EFI_LOCATE_PROTOCOL) (
    IN EFI_GUID   *Protocol,
    IN VOID       *Registration OPTIONAL,
    OUT VOID      **Interface
);  

Parameters
Protocol Provides the protocol to search for.
Registration Optional registration key returned from EFI_BOOT_SERVICES.RegisterProtocolNotify(). If Registration is NULL, then it is ignored.
Interface On return, a pointer to the first interface that matches Protocol and Registration.

Description
The LocateProtocol() function finds the first device handle that support Protocol, and returns a pointer to the protocol interface from that handle in Interface. If no protocol instances are found, then Interface is set to NULL.

If Interface is NULL, then EFI_INVALID_PARAMETER is returned.
If Protocol is NULL, then EFI_INVALID_PARAMETER is returned.

If Registration is NULL, and there are no handles in the handle database that support Protocol, then EFI_NOT_FOUND is returned.

If Registration is not NULL, and there are no new handles for Registration, then EFI_NOT_FOUND is returned.

### Status Codes Returned

<table>
<thead>
<tr>
<th>Status Code</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>EFI_SUCCESS</td>
<td>A protocol instance matching Protocol was found and returned in Interface.</td>
</tr>
<tr>
<td>EFI_INVALID_PARAMETER</td>
<td>Interface is NULL.</td>
</tr>
<tr>
<td></td>
<td>Protocol is NULL.</td>
</tr>
<tr>
<td>EFI_NOT_FOUND</td>
<td>No protocol instances were found that match Protocol and Registration.</td>
</tr>
</tbody>
</table>

---

### EFI_BOOT_SERVICES.InstallMultipleProtocolInterfaces()

#### Summary

Installs one or more protocol interfaces into the boot services environment.

#### Prototype

```c
typedef EFI_STATUS EFIAPI *EFI_INSTALL_MULTIPLE_PROTOCOL_INTERFACES) (  
    IN OUT EFI_HANDLE *Handle,
    ...
);
```

#### Parameters

- **Handle** The pointer to a handle to install the new protocol interfaces on, or a pointer to NULL if a new handle is to be allocated.
- **...** A variable argument list containing pairs of protocol GUIDs and protocol interfaces.

#### Description

This function installs a set of protocol interfaces into the boot services environment. It removes arguments from the variable argument list in pairs. The first item is always a pointer to the protocol’s GUID, and the second item is always a pointer to the protocol’s interface. These pairs are used to call the boot service EFI_BOOT_SERVICES.InstallProtocolInterface() to add a protocol interface to Handle. If Handle is NULL on entry, then a new handle will be allocated. The pairs of arguments are removed in order from the variable argument list until a NULL protocol GUID value is found. If any errors are generated while the protocol interfaces are being installed, then all the protocols installed prior to the error will be uninstalled with the boot service.
EFI_BOOT_SERVICES.UninstallProtocolInterface() before the error is returned. The same GUID cannot be installed more than once onto the same handle.

It is illegal to have two handles in the handle database with identical device paths. This service performs a test to guarantee a duplicate device path is not inadvertently installed on two different handles. Before any protocol interfaces are installed onto Handle, the list of GUID/pointer pair parameters are searched to see if a Device Path Protocol instance is being installed. If a Device Path Protocol instance is going to be installed onto Handle, then a check is made to see if a handle is already present in the handle database with an identical Device Path Protocol instance. If an identical Device Path Protocol instance is already present in the handle database, then no protocols are installed onto Handle, and EFI_ALREADY_STARTED is returned.

### Status Codes Returned

<table>
<thead>
<tr>
<th>Code</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>EFI_SUCCESS</td>
<td>All the protocol interfaces were installed.</td>
</tr>
<tr>
<td>EFI_ALREADY_STARTED</td>
<td>A Device Path Protocol instance was passed in that is already present in the handle database.</td>
</tr>
<tr>
<td>EFI_OUT_OF_RESOURCES</td>
<td>There was not enough memory in pool to install all the protocols.</td>
</tr>
<tr>
<td>EFI_INVALID_PARAMETER</td>
<td>Handle is NULL.</td>
</tr>
<tr>
<td>EFI_INVALID_PARAMETER</td>
<td>Protocol is already installed on the handle specified by Handle.</td>
</tr>
</tbody>
</table>

**EFI_BOOT_SERVICES.UninstallMultipleProtocolInterfaces()**

**Summary**

Removes one or more protocol interfaces into the boot services environment.

**Prototype**

```c
typedef
EFI_STATUS
EFIAPI *EFI_UNINSTALL_MULTIPLE_PROTOCOL_INTERFACES) (  
  IN EFI_HANDLE Handle,
  ...
);
```

**Parameters**

- `Handle` The handle to remove the protocol interfaces from.
- `...` A variable argument list containing pairs of protocol GUIDs and protocol interfaces.

**Description**

This function removes a set of protocol interfaces from the boot services environment. It removes arguments from the variable argument list in pairs. The first item is always a pointer to the protocol’s GUID, and the second item is always a pointer to the protocol’s interface. These pairs are used to call the boot service EFI_BOOT_SERVICES.UninstallProtocolInterface() to remove a protocol interface from Handle. The pairs of arguments are removed in order from the variable argument list until a NULL protocol GUID value is found. If all of the protocols are uninstalled from Handle, then
**EFI_SUCCESS** is returned. If any errors are generated while the protocol interfaces are being uninstalled, then the protocols uninstalled prior to the error will be reinstalled with the boot service **EFI_BOOT_SERVICES.InstallProtocolInterface()** and the status code **EFI_INVALID_PARAMETER** is returned.

### Status Codes Returned

<table>
<thead>
<tr>
<th>Status Code</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>EFI_SUCCESS</td>
<td>All the protocol interfaces were removed.</td>
</tr>
<tr>
<td>EFI_INVALID_PARAMETER</td>
<td>One of the protocol interfaces was not previously installed on Handle.</td>
</tr>
</tbody>
</table>

### 7.4 Image Services

Three types of images can be loaded: UEFI applications written (see Section 2.1.2), UEFI boot services drivers (see Section 2.1.4), and EFI runtime drivers (see Section 2.1.4). A UEFI OS Loader (see Section 2.1.3) is a type of UEFI application. The most significant difference between these image types is the type of memory into which they are loaded by the firmware’s loader. Table 7-8 summarizes the differences between images.

### Table 7-8 Image Type Differences Summary

<table>
<thead>
<tr>
<th></th>
<th>UEFI Application</th>
<th>UEFI Boot Service Driver</th>
<th>UEFI Runtime Driver</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Description</strong></td>
<td>A transient application that is loaded during boot services time. UEFI applications are either unloaded when they complete (see Section 2.1.2), or they take responsibility for the continued operation of the system via ExitBootServices() (see Section 2.1.3). The UEFI applications are loaded in sequential order by the boot manager, but one UEFI application may dynamically load another.</td>
<td>A program that is loaded into boot services memory and stays resident until boot services terminate. See Section 2.1.4.</td>
<td>A program that is loaded into runtime services memory and stays resident during runtime. The memory required for a UEFI runtime services driver must be performed in a single memory allocation, and marked as EfiRuntimeServicesData. (Note that the memory only stays resident when booting an EFI-compatible operating system. Legacy operating systems will reuse the memory,) See Section 2.1.4.</td>
</tr>
<tr>
<td><strong>Loaded into memory type</strong></td>
<td><strong>EfiLoaderCode, EfiLoaderData</strong></td>
<td><strong>EfiBootServicesCode, EfiBootServicesData</strong></td>
<td><strong>EfiRuntimeServicesCode, EfiRuntimeServicesData</strong></td>
</tr>
<tr>
<td><strong>Default pool allocations from memory type</strong></td>
<td>EfiLoaderData</td>
<td>EfiBootServicesData</td>
<td>EfiRuntimeServicesData</td>
</tr>
</tbody>
</table>
Most UEFI images are loaded by the boot manager. When a UEFI application or UEFI driver is installed, the installation procedure registers itself with the boot manager for loading. However, in some cases a UEFI application or UEFI driver may want to programmatically load and start another UEFI image. This can be done with the EFI_BOOT_SERVICES.LoadImage() and EFI_BOOT_SERVICES.StartImage() interfaces. UEFI drivers may only load UEFI applications during the UEFI driver’s initialization entry point. Table 7-9 lists the functions that make up Image Services.

**Table 7-9 Image Functions**

<table>
<thead>
<tr>
<th>Name</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>LoadImage</td>
<td>Boot</td>
<td>Loads an EFI image into memory.</td>
</tr>
<tr>
<td>StartImage</td>
<td>Boot</td>
<td>Transfers control to a loaded image’s entry point.</td>
</tr>
<tr>
<td>UnloadImage</td>
<td>Boot</td>
<td>Unloads an image.</td>
</tr>
<tr>
<td>EFI_IMAGE_ENTRY_POINT</td>
<td>Boot</td>
<td>Prototype of an EFI Image’s entry point.</td>
</tr>
<tr>
<td>Exit</td>
<td>Boot</td>
<td>Exits the image’s entry point.</td>
</tr>
<tr>
<td>ExitBootServices</td>
<td>Boot</td>
<td>Terminates boot services.</td>
</tr>
</tbody>
</table>

The Image boot services have been modified to take advantage of the information that is now being tracked with the EFI_BOOT_SERVICES.OpenProtocol() and EFI_BOOT_SERVICES.CloseProtocol() boot services. Since the usage of protocol interfaces is being tracked with these new boot services, it is now possible to automatically close protocol interfaces when a UEFI application or a UEFI driver is unloaded or exited.

**EFI_BOOT_SERVICES.LoadImage()**

**Summary**

Loads an EFI image into memory.

**Prototype**
typedef

EFI_STATUS

(EIFIAPI *EFI_IMAGE_LOAD) (  
    IN BOOLEAN BootPolicy,
    IN EFI_HANDLE ParentImageHandle,
    IN EFI_DEVICE_PATH_PROTOCOL *DevicePath,
    IN VOID *SourceBuffer OPTIONAL,
    IN UINTN SourceSize,
    OUT EFI_HANDLE *ImageHandle
);

Parameters

BootPolicy If TRUE, indicates that the request originates from the boot manager, and that the boot manager is attempting to load DevicePath as a boot selection. Ignored if SourceBuffer is not NULL.

ParentImageHandle The caller’s image handle. Type EFI_HANDLE is defined in the EFI_BOOT_SERVICES.InstallProtocolInterface() function description. This field is used to initialize the ParentHandle field of the EFI_LOADED_IMAGE_PROTOCOL for the image that is being loaded.

DevicePath The DeviceHandle specific file path from which the image is loaded. EFI_DEVICE_PATH_PROTOCOL is defined in Section 10.2.

SourceBuffer If not NULL, a pointer to the memory location containing a copy of the image to be loaded.

SourceSize The size in bytes of SourceBuffer. Ignored if SourceBuffer is NULL.

ImageHandle Pointer to the returned image handle that is created when the image is successfully loaded. Type EFI_HANDLE is defined in the InstallProtocolInterface() function description.

Related Definitions

#define EFI_HII_PACKAGE_LIST_PROTOCOL_GUID  
{ 0x6a1ee763, 0xd47a, 0x43b4,  
{ 0xaa, 0xbe, 0xef, 0x1d, 0xe2, 0xab, 0x56, 0xfc } }  

typedef EFI_HII_PACKAGE_LIST_HEADER *EFI_HII_PACKAGE_LIST_PROTOCOL;

Description

The LoadImage() function loads an EFI image into memory and returns a handle to the image. The image is loaded in one of two ways:

• If SourceBuffer is not NULL, the function is a memory-to-memory load in which SourceBuffer points to the image to be loaded and SourceSize indicates the image’s size in bytes. In this case, the caller has copied the image into SourceBuffer and can free the buffer once loading is complete.

• If SourceBuffer is NULL, the function is a file copy operation that uses the EFI_SIMPLE_FILE_SYSTEM_PROTOCOL.
If there is no instance of \texttt{EFI\_SIMPLE\_FILE\_SYSTEM\_PROTOCOL} associated with file path, then this function will attempt to use \texttt{EFI\_LOAD\_FILE\_PROTOCOL} (\texttt{BootPolicy} is \texttt{TRUE}) or \texttt{EFI\_LOAD\_FILE2\_PROTOCOL}, and then \texttt{EFI\_LOAD\_FILE\_PROTOCOL} (\texttt{BootPolicy} is \texttt{FALSE}).

In all cases, this function will use the instance of these protocols associated with the handle that most closely matches \texttt{DevicePath} will be used. See the boot service description for more information on how the closest handle is located.

- In the case of \texttt{EFI\_SIMPLE\_FILE\_SYSTEM\_PROTOCOL}, the path name from the File Path Media Device Path node(s) of \texttt{DevicePath} is used.
- In the case of \texttt{EFI\_LOAD\_FILE\_PROTOCOL}, the remaining device path nodes of \texttt{DevicePath} and the \texttt{BootPolicy} flag are passed to the \texttt{EFI\_LOAD\_FILE\_PROTOCOL.LoadFile()} function. The default image responsible for booting is loaded when \texttt{DevicePath} specifies only the device (and there are no further device nodes). For more information see the discussion of the \texttt{EFI\_LOAD\_FILE\_PROTOCOL} in Section 13.1.
- In the case of \texttt{EFI\_LOAD\_FILE2\_PROTOCOL}, the behavior is the same as above, except that it is only used if \texttt{BootOption} is \texttt{FALSE}. For more information, see the discussion of the \texttt{EFI\_LOAD\_FILE2\_PROTOCOL}.
- If the platform supports driver signing, as specified in Section 32.4.2, and the image signature is not valid, then information about the image is recorded in the \texttt{EFI\_IMAGE\_EXECUTION\_INFO\_TABLE} (see Section 32.4.2) and \texttt{EFI\_SECURITY\_VIOLATION} is returned.
- If the platform supports user authentication, as described in Section 36, and loading of images on the specified \texttt{FilePath} is forbidden in the current user profile, then the information about the image is recorded (see Deferred Execution in Section 36.1.5) and \texttt{EFI\_SECURITY\_VIOLATION} is returned.

Once the image is loaded, firmware creates and returns an \texttt{EFI\_HANDLE} that identifies the image and supports \texttt{EFI\_LOADED\_IMAGE\_PROTOCOL} and the \texttt{EFI\_LOADED\_IMAGE\_DEVICE\_PATH\_PROTOCOL}. The caller may fill in the image’s “load options” data, or add additional protocol support to the handle before passing control to the newly loaded image by calling \texttt{EFI\_BOOT\_SERVICES.StartImage()}. Also, once the image is loaded, the caller either starts it by calling \texttt{StartImage()} or unloads it by calling \texttt{EFI\_BOOT\_SERVICES.UnloadImage()}. Once the image is loaded, \texttt{LoadImage()} installs \texttt{EFI\_HII\_PACKAGE\_LIST\_PROTOCOL} on the handle if the image contains a custom PE/COFF resource with the type 'HII'. The protocol's interface pointer points to the HII package list which is contained in the resource's data. The format of this is in Section 33.3.1.
Status Codes Returned

<table>
<thead>
<tr>
<th>Code</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>EFI_SUCCESS</td>
<td>Image was loaded into memory correctly.</td>
</tr>
<tr>
<td>EFI_NOT_FOUND</td>
<td>Both SourceBuffer and DevicePath are NULL.</td>
</tr>
<tr>
<td>EFI_INVALID_PARAMETER</td>
<td>One of the parameters has an invalid value.</td>
</tr>
<tr>
<td>EFI_INVALID_PARAMETER</td>
<td>ImageHandle is NULL.</td>
</tr>
<tr>
<td>EFI_INVALID_PARAMETER</td>
<td>ParentImageHandle is NULL.</td>
</tr>
<tr>
<td>EFI_OUT_OF_RESOURCES</td>
<td>Image was not loaded due to insufficient resources.</td>
</tr>
<tr>
<td>EFI_LOAD_ERROR</td>
<td>Image was not loaded because the image format was corrupt or not understood.</td>
</tr>
<tr>
<td>EFI_DEVICE_ERROR</td>
<td>Image was not loaded because the device returned a read error.</td>
</tr>
<tr>
<td>EFI_ACCESS_DENIED</td>
<td>Image was not loaded because the platform policy prohibits the image from being loaded. NULL is returned in *ImageHandle.</td>
</tr>
<tr>
<td>EFI_SECURITY_VIOLATION</td>
<td>Image was loaded and an ImageHandle was created with a valid EPILE_LOADED_IMAGE_PROTOCOL. However, the current platform policy specifies that the image should not be started.</td>
</tr>
</tbody>
</table>

**EFI_BOOT_SERVICES.StartImage()**

**Summary**
Transfers control to a loaded image’s entry point.

**Prototype**

```c
typedef EFI_STATUS
    (EFIAPI *EFI_IMAGE_START) (
    IN EFI_HANDLE ImageHandle,
    OUT UINTN *ExitDataSize,
    OUT CHAR16 **ExitData OPTIONAL
    );
```

**Parameters**

- **ImageHandle**
  Handle of image to be started. Type EFI_HANDLE is defined in the EFI_BOOT_SERVICES.InstallProtocolInterface() function description.

- **ExitDataSize**
  Pointer to the size, in bytes, of ExitData. If ExitData is NULL, then this parameter is ignored and the contents of ExitDataSize are not modified.

- **ExitData**
  Pointer to a pointer to a data buffer that includes a Null-terminated string, optionally followed by additional binary data. The string is a description that the caller may use to further indicate the reason for the image’s exit.
Description

The \texttt{StartImage()} function transfers control to the entry point of an image that was loaded by \texttt{EFI\_BOOT\_SERVICES.LoadImage()}. The image may only be started one time.

Control returns from \texttt{StartImage()} when the loaded image’s \texttt{EFI\_IMAGE\_ENTRY\_POINT} returns or when the loaded image calls \texttt{EFI\_BOOT\_SERVICES.Exit()}. When that call is made, the \texttt{ExitData} buffer and \texttt{ExitDataSize} from \texttt{Exit()} are passed back through the \texttt{ExitData} buffer and \texttt{ExitDataSize} in this function. The caller of this function is responsible for returning the \texttt{ExitData} buffer to the pool by calling \texttt{EFI\_BOOT\_SERVICES.FreePool()} when the buffer is no longer needed. Using \texttt{Exit()} is similar to returning from the image’s \texttt{EFI\_IMAGE\_ENTRY\_POINT} except that \texttt{Exit()} may also return additional \texttt{ExitData}. \texttt{Exit()} function description defines clean up procedure performed by the firmware once loaded image returns control.

\textbf{EFI 1.10 Extension}

To maintain compatibility with UEFI drivers that are written to the \textit{EFI 1.02 Specification}, \texttt{StartImage()} must monitor the handle database before and after each image is started. If any handles are created or modified when an image is started, then \texttt{EFI\_BOOT\_SERVICES.ConnectController()} must be called with the \texttt{Recursive} parameter set to \texttt{TRUE} for each of the newly created or modified handles before \texttt{StartImage()} returns.

\textbf{Status Codes Returned}

<table>
<thead>
<tr>
<th>Code</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>\texttt{EFI_INVALID_PARAMETER}</td>
<td>\texttt{ImageHandle} is either an invalid image handle or the image has already been initialized with StartImage.</td>
</tr>
<tr>
<td>Exit code from image</td>
<td>Exit code from image.</td>
</tr>
<tr>
<td>\texttt{EFI_SECURITY_VIOLATION}</td>
<td>The current platform policy specifies that the image should not be started.</td>
</tr>
</tbody>
</table>

\textbf{EFI\_BOOT\_SERVICES.UnloadImage()}

\textbf{Summary}

Unloads an image.

\textbf{Prototype}

\begin{verbatim}
typedef
  EFI_STATUS
  (EFIAPI *EFI\_IMAGE\_UNLOAD) (  
    IN EFI\_HANDLE     ImageHandle 
  );
\end{verbatim}

\textbf{Parameters}

\begin{itemize}
  \item \texttt{ImageHandle} Handle that identifies the image to be unloaded.
\end{itemize}

\textbf{Description}

The \texttt{UnloadImage()} function unloads a previously loaded image.

There are three possible scenarios. If the image has not been started, the function unloads the image and returns \texttt{EFI\_SUCCESS}. 
If the image has been started and has an `Unload()` entry point, control is passed to that entry point. If the image’s unload function returns `EFI_SUCCESS`, the image is unloaded; otherwise, the error returned by the image’s unload function is returned to the caller. The image unload function is responsible for freeing all allocated memory and ensuring that there are no references to any freed memory, or to the image itself, before returning `EFI_SUCCESS`.

If the image has been started and does not have an `Unload()` entry point, the function returns `EFI_UNSUPPORTED`.

**EFI 1.10 Extension**

All of the protocols that were opened by `ImageHandle` using the boot service `EFI_BOOT_SERVICES.OpenProtocol()` are automatically closed with the boot service `EFI_BOOT_SERVICES.CloseProtocol()`. If all of the open protocols are closed, then `EFI_SUCCESS` is returned. If any call to `CloseProtocol()` fails, then the error code from `CloseProtocol()` is returned.

**Status Codes Returned**

<table>
<thead>
<tr>
<th>Code</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>EFI_SUCCESS</code></td>
<td>The image has been unloaded.</td>
</tr>
<tr>
<td><code>EFI_UNSUPPORTED</code></td>
<td>The image has been started, and does not support unload.</td>
</tr>
<tr>
<td><code>EFI_INVALID_PARAMETER</code></td>
<td><code>ImageHandle</code> is not a valid image handle.</td>
</tr>
<tr>
<td>Exit code from Unload handler</td>
<td>Exit code from the image’s unload function.</td>
</tr>
</tbody>
</table>

**EFI_IMAGE_ENTRY_POINT**

**Summary**

This is the declaration of an EFI image entry point. This can be the entry point to an application written to this specification, an EFI boot service driver, or an EFI runtime driver.

**Prototype**

```c
typedef EFI_STATUS (EFIAPIC *EFI_IMAGE_ENTRY_POINT) (  
    IN EFI_HANDLE ImageHandle,  
    IN EFI_SYSTEM_TABLE *SystemTable  
);
```

**Parameters**

- **ImageHandle**  
  Handle that identifies the loaded image. Type `EFI_HANDLE` is defined in the `EFI_BOOT_SERVICES.InstallProtocolInterface()` function description.

- **SystemTable**  
  System Table for this image. Type `EFI_SYSTEM_TABLE` is defined in [Section 4](#).

---

EFI Forum, Inc.  
March 2021  
222
**Description**

An image’s entry point is of type `EFI_IMAGE_ENTRY_POINT`. After firmware loads an image into memory, control is passed to the image’s entry point. The entry point is responsible for initializing the image. The image’s `ImageHandle` is passed to the image. The `ImageHandle` provides the image with all the binding and data information it needs. This information is available through protocol interfaces. However, to access the protocol interfaces on `ImageHandle` requires access to boot services functions. Therefore, `EFI_BOOT_SERVICES.LoadImage()` passes to the `EFI_IMAGE_ENTRY_POINT` a `SystemTable` that is inherited from the current scope of `LoadImage()`.

All image handles support the `EFI_LOADED_IMAGE_PROTOCOL` and the `EFI_LOADED_IMAGE_DEVICE_PATH_PROTOCOL`. These protocol can be used to obtain information about the loaded image’s state—for example, the device from which the image was loaded and the image’s load options. In addition, the `ImageHandle` may support other protocols provided by the parent image.

If the image supports dynamic unloading, it must supply an unload function in the `EFI_LOADED_IMAGE_PROTOCOL` structure before returning control from its entry point.

In general, an image returns control from its initialization entry point by calling `EFI_BOOT_SERVICES.Exit()` or by returning control from its entry point. If the image returns control from its entry point, the firmware passes control to `Exit()` using the return code as the `ExitStatus` parameter to `Exit()`.

See `Exit()` below for entry point exit conditions.

**EFI_BOOT_SERVICES.Exit()**

**Summary**

Terminates a loaded EFI image and returns control to boot services.

**Prototype**

```c
typedef EFI_STATUS
 (EFIAPI *EFI_EXIT) (
  IN EFI_HANDLE ImageHandle,
  IN EFI_STATUS ExitStatus,
  IN UINTN ExitDataSize,
  IN CHAR16 *ExitData OPTIONAL
 );
```

**Parameters**

- `ImageHandle` Handle that identifies the image. This parameter is passed to the image on entry.
- `ExitStatus` The image’s exit code.
- `ExitDataSize` The size, in bytes, of `ExitData`. Ignored if `ExitStatus` is `EFI_SUCCESS`.
- `ExitData` Pointer to a data buffer that includes a Null-terminated string, optionally followed by additional binary data. The string is a
description that the caller may use to further indicate the reason for the image’s exit. \textit{ExitData} is only valid if \textit{ExitStatus} is something other than \texttt{EFI\_SUCCESS}. The \textit{ExitData} buffer must be allocated by calling \texttt{EFI\_BOOT\_SERVICES.AllocatePool()}.

**Description**

The \texttt{Exit()} function terminates the image referenced by \textit{ImageHandle} and returns control to boot services. This function may not be called if the image has already returned from its entry point (\texttt{EFI\_IMAGE\_ENTRY\_POINT}) or if it has loaded any child images that have not exited (all child images must exit before this image can exit).

Using \texttt{Exit()} is similar to returning from the image’s \texttt{EFI\_IMAGE\_ENTRY\_POINT} except that \texttt{Exit()} may also return additional \textit{ExitData}.

When an application exits a compliant system, firmware frees the memory used to hold the image. The firmware also frees its references to the \textit{ImageHandle} and the handle itself. Before exiting, the application is responsible for freeing any resources it allocated. This includes memory (pages and/or pool), open file system handles, and so forth. The only exception to this rule is the \textit{ExitData} buffer, which must be freed by the caller of \texttt{EFI\_BOOT\_SERVICES.StartImage()}. (If the buffer is needed, firmware must allocate it by calling \texttt{EFI\_BOOT\_SERVICES.AllocatePool()} and must return a pointer to it to the caller of \texttt{StartImage()}.)

When an EFI boot service driver or runtime service driver exits, firmware frees the image only if the \textit{ExitStatus} is an error code; otherwise the image stays resident in memory. The driver must not return an error code if it has installed any protocol handlers or other active callbacks into the system that have not (or cannot) be cleaned up. If the driver exits with an error code, it is responsible for freeing all resources before exiting. This includes any allocated memory (pages and/or pool), open file system handles, and so forth.

It is valid to call \texttt{Exit()} or \texttt{UnloadImage()} for an image that was loaded by \texttt{EFI\_BOOT\_SERVICES.LoadImage()} before calling \texttt{EFI\_BOOT\_SERVICES.StartImage()}. This will free the image from memory without having started it.

**EFI 1.10 Extension**

If \textit{ImageHandle} is a UEFI application, then all of the protocols that were opened by \textit{ImageHandle} using the boot service \texttt{EFI\_BOOT\_SERVICES.OpenProtocol()} are automatically closed with the boot service \texttt{EFI\_BOOT\_SERVICES.CloseProtocol()}. If \textit{ImageHandle} is a UEFI boot service driver or UEFI runtime service driver, and \textit{ExitStatus} is an error code, then all of the protocols that were opened by \textit{ImageHandle} using the boot service \texttt{OpenProtocol()} are automatically closed with the boot service \texttt{CloseProtocol()}. If \textit{ImageHandle} is a UEFI boot service driver or UEFI runtime service driver, and \textit{ExitStatus} is not an error code, then no protocols are automatically closed by this service.
Status Codes Returned

<table>
<thead>
<tr>
<th>Code</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>(Does not return.)</td>
<td>Image exit. Control is returned to the <code>StartImage()</code> call that invoked the image specified by <code>ImageHandle</code>.</td>
</tr>
<tr>
<td>EFI_SUCCESS</td>
<td>The image specified by <code>ImageHandle</code> was unloaded. This condition only occurs for images that have been loaded with <code>LoadImage()</code> but have not been started with <code>StartImage()</code>.</td>
</tr>
<tr>
<td>EFI_INVALID_PARAMETER</td>
<td>The image specified by <code>ImageHandle</code> has been loaded and started with <code>LoadImage()</code> and <code>StartImage()</code>, but the image is not the currently executing image.</td>
</tr>
</tbody>
</table>

**EFI_BOOT_SERVICES.ExitBootServices()**

Summary
Terminates all boot services.

Prototype

```c
typedef EFI_STATUS
(EFIAPI *EFI_BOOT_SERVICES)(
    IN EFI_HANDLE ImageHandle,
    IN UINTN     MapKey
);
```

Parameters

- `ImageHandle` Handle that identifies the exiting image. Type `EFI_HANDLE` is defined in the EFI_BOOT_SERVICES.InstallProtocolInterface() function description.
- `MapKey` Key to the latest memory map.

Description
The `ExitBootServices()` function is called by the currently executing UEFI OS loader image to terminate all boot services. On success, the UEFI OS loader becomes responsible for the continued operation of the system. All events from the EFI_EVENT_GROUP_BEFORE_EXIT_BOOT_SERVICES and EFI_EVENT_GROUP_EXIT_BOOT_SERVICES event notification groups as well as events of type EVT_SIGNAL_EXIT_BOOT_SERVICES must be signaled before `ExitBootServices()` returns `EFI_SUCCESS`. The events are only signaled once even if `ExitBootServices()` is called multiple times.

A UEFI OS loader must ensure that it has the system’s current memory map at the time it calls `ExitBootServices()`. This is done by passing in the current memory map’s `MapKey` value as returned by EFI_BOOT_SERVICES.GetMemoryMap(). Care must be taken to ensure that the memory map does not change between these two calls. It is suggested that `GetMemoryMap()` be called immediately before calling `ExitBootServices()`. If `MapKey` value is incorrect, `ExitBootServices()` returns `EFI_INVALID_PARAMETER` and `GetMemoryMap()` with `ExitBootServices()` must be called again. Firmware implementation may choose to do a partial shutdown of the boot services during the first call.
to ExitBootServices(). A UEFI OS loader should not make calls to any boot service function other than Memory Allocation Services after the first call to ExitBootServices().

On success, the UEFI OS loader owns all available memory in the system. In addition, the UEFI OS loader can treat all memory in the map marked as EfiBootServicesCode and EfiBootServicesData as available free memory. No further calls to boot service functions or EFI device-handle-based protocols may be used, and the boot services watchdog timer is disabled. On success, several fields of the EFI System Table should be set to NULL. These include ConsoleInHandle, ConIn, ConsoleOutHandle, ConOut, StandardErrorHandle, StdErr, and BootServicesTable. In addition, since fields of the EFI System Table are being modified, the 32-bit CRC for the EFI System Table must be recomputed.

**Firmware must guarantee the following order of processing:**
- EFI_EVENT_GROUP_BEFORE_EXIT_BOOT_SERVICES handlers are called.
- Timer services are deactivated (timer event activity is stopped).
- EVT_SIGNAL_EXIT_BOOT_SERVICES and EFI_EVENT_GROUP_EXIT_BOOT_SERVICES handlers are called.

**NOTE:** The EVT_SIGNAL_EXIT_BOOT_SERVICES event type and EFI_EVENT_GROUP_EXIT_BOOT_SERVICES event group are functionally equivalent. Firmware does not distinguish between the two while ordering the handlers.

Refer to EFI_EVENT_GROUP_EXIT_BOOT_SERVICES description in the EFI_BOOT_SERVICES.CreateEventEx() section above for additional restrictions on EXIT_BOOT_SERVICES handlers.

**Status Codes Returned**

<table>
<thead>
<tr>
<th>Code</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>EFI_SUCCESS</td>
<td>Boot services have been terminated.</td>
</tr>
<tr>
<td>EFI_INVALID_PARAMETER</td>
<td>MapKey is incorrect.</td>
</tr>
</tbody>
</table>

### 7.5 Miscellaneous Boot Services

This section contains the remaining function definitions for boot services not defined elsewhere but which are required to complete the definition of the EFI environment. Table 7-10 lists the Miscellaneous Boot Services Functions.

<table>
<thead>
<tr>
<th>Name</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>SetWatchDogTimer</td>
<td>Boot</td>
<td>Resets and sets a watchdog timer used during boot services time.</td>
</tr>
<tr>
<td>Stall</td>
<td>Boot</td>
<td>Stalls the processor.</td>
</tr>
<tr>
<td>CopyMem</td>
<td>Boot</td>
<td>Copies the contents of one buffer to another buffer.</td>
</tr>
<tr>
<td>SetMem</td>
<td>Boot</td>
<td>Fills a buffer with a specified value.</td>
</tr>
<tr>
<td>GetNextMonotonicCount</td>
<td>Boot</td>
<td>Returns a monotonically increasing count for the platform.</td>
</tr>
<tr>
<td>InstallConfigurationTable</td>
<td>Boot</td>
<td>Adds, updates, or removes a configuration table from the EFI System Table.</td>
</tr>
<tr>
<td>CalculateCrc32</td>
<td>Boot</td>
<td>Computes and returns a 32-bit CRC for a data buffer.</td>
</tr>
</tbody>
</table>
The EFI_BOOT_SERVICES.CalculateCrc32() service was added because there are several places in EFI that 32-bit CRCs are used. These include the EFI System Table, the EFI Boot Services Table, the EFI Runtime Services Table, and the GUID Partition Table (GPT) structures. The CalculateCrc32() service allows new 32-bit CRCs to be computed, and existing 32-bit CRCs to be validated.

EFI_BOOT_SERVICES.SetWatchdogTimer()

Summary
Sets the system’s watchdog timer.

Prototype

typedef
    EFI_STATUS
    (EFI_API *EFI_SET_WATCHDOG_TIMER) ((
        IN UINTN   Timeout,
        IN UINT64  WatchdogCode,
        IN UINTN   DataSize,
        IN CHAR16  *WatchdogData OPTIONAL
    );

Parameters

Timeout The number of seconds to set the watchdog timer to. A value of zero disables the timer.

WatchdogCode The numeric code to log on a watchdog timer timeout event. The firmware reserves codes 0x0000 to 0xFFFF. Loaders and operating systems may use other timeout codes.

DataSize The size, in bytes, of WatchdogData.

WatchdogData A data buffer that includes a Null-terminated string, optionally followed by additional binary data. The string is a description that the call may use to further indicate the reason to be logged with a watchdog event.

Description

The SetWatchdogTimer() function sets the system’s watchdog timer.

If the watchdog timer expires, the event is logged by the firmware. The system may then either reset with the Runtime Service ResetSystem(), or perform a platform specific action that must eventually cause the platform to be reset. The watchdog timer is armed before the firmware’s boot manager invokes an EFI boot option. The watchdog must be set to a period of 5 minutes. The EFI Image may reset or disable the watchdog timer as needed. If control is returned to the firmware’s boot manager, the watchdog timer must be disabled.

The watchdog timer is only used during boot services. On successful completion of EFI_BOOT_SERVICES.ExitBootServices() the watchdog timer is disabled.

The accuracy of the watchdog timer is +/- 1 second from the requested Timeout.
Status Codes Returned

<table>
<thead>
<tr>
<th>Status Code</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>EFI_SUCCESS</td>
<td>The timeout has been set.</td>
</tr>
<tr>
<td>EFI_INVALID_PARAMETER</td>
<td>The supplied WatchdogCode is invalid.</td>
</tr>
<tr>
<td>EFI_UNSUPPORTED</td>
<td>The system does not have a watchdog timer.</td>
</tr>
<tr>
<td>EFI_DEVICE_ERROR</td>
<td>The watch dog timer could not be programmed due to a hardware error.</td>
</tr>
</tbody>
</table>

EFI_BOOT_SERVICES.Stall()

Summary
Induces a fine-grained stall.

Prototype

```c
typedef
EFI_STATUS
(EIFIAPI *EFISTALL)(
    IN UINTN    Microseconds
);
```

Parameters

- **Microseconds**
  The number of microseconds to stall execution.

Description

The Stall() function stalls execution on the processor for at least the requested number of microseconds. Execution of the processor is not yielded for the duration of the stall.

Status Codes Returned

<table>
<thead>
<tr>
<th>Status Code</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>EFI_SUCCESS</td>
<td>Execution was stalled at least the requested number of Microseconds.</td>
</tr>
</tbody>
</table>

EFI_BOOT_SERVICES.CopyMem()

Summary

The CopyMem() function copies the contents of one buffer to another buffer.
Prototype

```c
typedef VOID
(EFIAPI *EFI_COPY_MEM) (
  IN VOID *Destination,
  IN VOID *Source,
  IN UINTN Length
);
```

Parameters

- `Destination` Pointer to the destination buffer of the memory copy.
- `Source` Pointer to the source buffer of the memory copy.
- `Length` Number of bytes to copy from `Source` to `Destination`.

Description

The `CopyMem()` function copies `Length` bytes from the buffer `Source` to the buffer `Destination`.

The implementation of `CopyMem()` must be reentrant, and it must handle overlapping `Source` and `Destination` buffers. This means that the implementation of `CopyMem()` must choose the correct direction of the copy operation based on the type of overlap that exists between the `Source` and `Destination` buffers. If either the `Source` buffer or the `Destination` buffer crosses the top of the processor’s address space, then the result of the copy operation is unpredictable.

The contents of the `Destination` buffer on exit from this service must match the contents of the `Source` buffer on entry to this service. Due to potential overlaps, the contents of the `Source` buffer may be modified by this service. The following rules can be used to guarantee the correct behavior:

1. If `Destination` and `Source` are identical, then no operation should be performed.
2. If `Destination` > `Source` and `Destination` < (`Source` + `Length`), then the data should be copied from the `Source` buffer to the `Destination` buffer starting from the end of the buffers and working toward the beginning of the buffers.
3. Otherwise, the data should be copied from the `Source` buffer to the `Destination` buffer starting from the beginning of the buffers and working toward the end of the buffers.

Status Codes Returned

None.

`EFI_BOOT_SERVICES.SetMem()`

Summary

The `SetMem()` function fills a buffer with a specified value.
Prototype

```c
typedef VOID EFIAPI *EFI_SET_MEM(
    IN VOID *Buffer,
    IN UINTN Size,
    IN UINT8 Value
);
```

Parameters

- **Buffer**
  - Pointer to the buffer to fill.
- **Size**
  - Number of bytes in Buffer to fill.
- **Value**
  - Value to fill Buffer with.

Description

This function fills *Size* bytes of *Buffer* with *Value*. The implementation of *SetMem()* must be reentrant. If *Buffer* crosses the top of the processor’s address space, the result of the *SetMem()* operation is unpredictable.

Status Codes Returned

None.

**EFI_BOOT_SERVICES.GetNextMonotonicCount()**

**Summary**

Returns a monotonically increasing count for the platform.

Prototype

```c
typedef EFI_STATUS (EFIAPI *EFI_GET_NEXT_MONOTONIC_COUNT)(
    OUT UINT64 *Count
);
```

Parameters

- **Count**
  - Pointer to returned value.

Description

The *GetNextMonotonicCount()* function returns a 64-bit value that is numerically larger than the last time the function was called.

The platform’s monotonic counter is comprised of two parts: the high 32 bits and the low 32 bits. The low 32-bit value is volatile and is reset to zero on every system reset. It is increased by 1 on every call to *GetNextMonotonicCount()*.

The high 32-bit value is nonvolatile and is increased by one on whenever the system resets or the low 32-bit counter overflows.
### Status Codes Returned

<table>
<thead>
<tr>
<th>Code</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>EFI_SUCCESS</td>
<td>The next monotonic count was returned.</td>
</tr>
<tr>
<td>EFI_DEVICE_ERROR</td>
<td>The device is not functioning properly.</td>
</tr>
<tr>
<td>EFI_INVALID_PARAMETER</td>
<td>Count is NULL.</td>
</tr>
</tbody>
</table>

### EFI_BOOT_SERVICES.InstallConfigurationTable()

#### Summary

Adds, updates, or removes a configuration table entry from the EFI System Table.

#### Prototype

```c
typedef EFI_STATUS
(EFIAPIC *EFI_INSTALL_CONFIGURATION_TABLE) (  
    IN EFI_GUID  *Guid,  
    IN VOID      *Table
);
```

#### Parameters

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Guid</td>
<td>A pointer to the GUID for the entry to add, update, or remove.</td>
</tr>
<tr>
<td>Table</td>
<td>A pointer to the configuration table for the entry to add, update, or remove. May be NULL.</td>
</tr>
</tbody>
</table>

#### Description

The `InstallConfigurationTable()` function is used to maintain the list of configuration tables that are stored in the EFI System Table. The list is stored as an array of (GUID, Pointer) pairs. The list must be allocated from pool memory with PoolType set to `EfiRuntimeServicesData`.

If `Guid` is NULL, `EFI_INVALID_PARAMETER` is returned. If `Guid` is valid, there are four possibilities:

- If `Guid` is not present in the System Table, and `Table` is not NULL, then the `(Guid, Table)` pair is added to the System Table. See Note below.
- If `Guid` is not present in the System Table, and `Table` is NULL, then `EFI_NOT_FOUND` is returned.
- If `Guid` is present in the System Table, and `Table` is not NULL, then the `(Guid, Table)` pair is updated with the new `Table` value.
- If `Guid` is present in the System Table, and `Table` is NULL, then the entry associated with `Guid` is removed from the System Table.

If an add, modify, or remove operation is completed, then `EFI_SUCCESS` is returned.

**Note:** If there is not enough memory to perform an add operation, then `EFI_OUT_OF_RESOURCES` is returned.
Status Codes Returned

<table>
<thead>
<tr>
<th>Code</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>EFI_SUCCESS</td>
<td>The (Guid, Table) pair was added, updated, or removed.</td>
</tr>
<tr>
<td>EFI_INVALID_PARAMETER</td>
<td>Guid is NULL.</td>
</tr>
<tr>
<td>EFI_NOT_FOUND</td>
<td>An attempt was made to delete a nonexistent entry.</td>
</tr>
<tr>
<td>EFI_OUT_OF_RESOURCES</td>
<td>There is not enough memory available to complete the operation.</td>
</tr>
</tbody>
</table>

**EFI_BOOT_SERVICES.CalculateCrc32()**

**Summary**
Computes and returns a 32-bit CRC for a data buffer.

**Prototype**

```c
typedef
EFI_STATUS
  (EFIAPIC *EFI_CALCULATE_CRC32)
IN VOID *Data,
IN UINTN DataSize,
OUT UINT32 *Crc32
);
```

**Parameters**

- **Data**
  A pointer to the buffer on which the 32-bit CRC is to be computed.

- **DataSize**
  The number of bytes in the buffer Data.

- **Crc32**
  The 32-bit CRC that was computed for the data buffer specified by Data and DataSize.

**Description**
This function computes the 32-bit CRC for the data buffer specified by Data and DataSize. If the 32-bit CRC is computed, then it is returned in Crc32 and EFI_SUCCESS is returned.

If Data is NULL, then EFI_INVALID_PARAMETER is returned.

If Crc32 is NULL, then EFI_INVALID_PARAMETER is returned.

If DataSize is 0, then EFI_INVALID_PARAMETER is returned.

**Status Codes Returned**

<table>
<thead>
<tr>
<th>Code</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>EFI_SUCCESS</td>
<td>The 32-bit CRC was computed for the data buffer and returned in Crc32.</td>
</tr>
<tr>
<td>EFI_INVALID_PARAMETER</td>
<td>Data is NULL.</td>
</tr>
<tr>
<td>EFI_INVALID_PARAMETER</td>
<td>Crc32 is NULL.</td>
</tr>
<tr>
<td>EFI_INVALID_PARAMETER</td>
<td>DataSize is 0.</td>
</tr>
</tbody>
</table>
8 - Services — Runtime Services

This section discusses the fundamental services that are present in a compliant system. The services are defined by interface functions that may be used by code running in the EFI environment. Such code may include protocols that manage device access or extend platform capability, as well as applications running in the preboot environment and EFI OS loaders.

Two types of services are described here:

- **Boot Services.** Functions that are available before a successful call to `EFI_BOOT_SERVICES.ExitBootServices()`. These functions are described in Section 7.

- **Runtime Services.** Functions that are available before and after any call to `ExitBootServices()`. These functions are described in this section.

During boot, system resources are owned by the firmware and are controlled through boot services interface functions. These functions can be characterized as “global” or “handle-based.” The term “global” simply means that a function accesses system services and is available on all platforms (since all platforms support all system services). The term “handle-based” means that the function accesses a specific device or device functionality and may not be available on some platforms (since some devices are not available on some platforms). Protocols are created dynamically. This section discusses the “global” functions and runtime functions; subsequent sections discuss the “handle-based.”

UEFI applications (including UEFI OS loaders) must use boot services functions to access devices and allocate memory. On entry, an image is provided a pointer to a system table which contains the Boot Services dispatch table and the default handles for accessing the console. All boot services functionality is available until a UEFI OS loader loads enough of its own environment to take control of the system’s continued operation and then terminates boot services with a call to `ExitBootServices()`.

In principle, the `ExitBootServices()` call is intended for use by the operating system to indicate that its loader is ready to assume control of the platform and all platform resource management. Thus boot services are available up to this point to assist the UEFI OS loader in preparing to boot the operating system. Once the UEFI OS loader takes control of the system and completes the operating system boot process, only runtime services may be called. Code other than the UEFI OS loader, however, may or may not choose to call `ExitBootServices()`. This choice may in part depend upon whether or not such code is designed to make continued use of EFI boot services or the boot services environment.

The rest of this section discusses individual functions. Runtime Services fall into these categories:

- Runtime Rules and Restrictions (Section 8.1)
- Variable Services (Section 8.1.1)
- Time Services (Section 8.3)
- Virtual Memory Services (Section 8.4)
- Miscellaneous Services (Section 8.5)

8.1 Runtime Services Rules and Restrictions

All of the Runtime Services may be called with interrupts enabled if desired. The Runtime Service functions will internally disable interrupts when it is required to protect access to hardware resources.
The interrupt enable control bit will be returned to its entry state after the access to the critical hardware resources is complete.

All callers of Runtime Services are restricted from calling the same or certain other Runtime Service functions prior to the completion and return of a previous Runtime Service call. These restrictions apply to:

- Runtime Services that have been interrupted
- Runtime Services that are active on another processor.

Callers are prohibited from using certain other services from another processor or on the same processor following an interrupt as specified in Table 8-1. For this table ‘Busy’ is defined as the state when a Runtime Service has been entered and has not returned to the caller.

The consequence of a caller violating these restrictions is undefined except for certain special cases described below.

### Table 8-1 Rules for Reentry Into Runtime Services

<table>
<thead>
<tr>
<th>If previous call is busy in</th>
<th>Forbidden to call</th>
</tr>
</thead>
<tbody>
<tr>
<td>Any</td>
<td>SetVirtualAddressMap()</td>
</tr>
<tr>
<td>ConvertPointer()</td>
<td>ConvertPointer()</td>
</tr>
<tr>
<td>SetVariable(),</td>
<td></td>
</tr>
<tr>
<td>UpdateCapsule(),</td>
<td></td>
</tr>
<tr>
<td>SetTime()</td>
<td></td>
</tr>
<tr>
<td>SetWakeupTime(),</td>
<td></td>
</tr>
<tr>
<td>GetNextHighMonotonicCount()</td>
<td></td>
</tr>
<tr>
<td>GetVariable()</td>
<td></td>
</tr>
<tr>
<td>GetNextVariableName()</td>
<td></td>
</tr>
<tr>
<td>SetVariable()</td>
<td></td>
</tr>
<tr>
<td>QueryVariableInfo()</td>
<td></td>
</tr>
<tr>
<td>UpdateCapsule()</td>
<td></td>
</tr>
<tr>
<td>QueryCapsuleCapabilities()</td>
<td></td>
</tr>
<tr>
<td>GetNextHighMonotonicCount()</td>
<td></td>
</tr>
<tr>
<td>GetTime()</td>
<td></td>
</tr>
<tr>
<td>SetTime()</td>
<td></td>
</tr>
<tr>
<td>GetWakeupTime()</td>
<td></td>
</tr>
<tr>
<td>SetWakeupTime()</td>
<td></td>
</tr>
</tbody>
</table>

The consequence of a caller violating these restrictions is undefined except for certain special cases described below.
If any `EFI_RUNTIME_SERVICES` calls are not supported for use by the OS at runtime, an EFI_RT_PROPERTIES_TABLE configuration table should be published describing which runtime services are supported at runtime (see Section 4.6). Note that this is merely a hint to the OS, which it is free to ignore, and so the platform is still required to provide callable implementations of unsupported runtime services that simply return EFI_UNSUPPORTED.

### 8.1.1 Exception for Machine Check, INIT, and NMI

Certain asynchronous events (e.g., NMI on IA-32 and x64 systems, Machine Check and INIT on Itanium systems) can not be masked and may occur with any setting of interrupt enabled. These events also may require OS level handler's involvement that may involve the invocation of some of the runtime services (see below).

If `SetVirtualAddressMap()` has been called, all calls to runtime services after Machine Check, INIT, or NMI, must be made using the virtual address map set by that call.

A Machine Check may have interrupted a runtime service (see below). If the OS determines that the Machine Check is recoverable, the OS level handler must follow the normal restrictions in Table 8-1.

If the OS determines that the Machine Check is non-recoverable, the OS level handler may ignore the normal restrictions and may invoke the runtime services described in Table 8-2 even in the case where a previous call was busy. The system firmware will honor the new runtime service call(s) and the operation of the previous interrupted call is not guaranteed. Any interrupted runtime functions will not be restarted.

The INIT and NMI events follow the same restrictions.

**Note:** On Itanium systems, the OS Machine Check Handler must not call `ResetSystem()`. If a reset is required, the OS Machine Check Handler may request SAL to reset upon return to SAL_CHECK.

The platform implementations are required to clear any runtime services in progress in order to enable the OS handler to invoke these runtime services even in the case where a previous call was busy. In this case, the proper operation of the original interrupted call is not guaranteed.

#### Table 8-2 Functions that may be called after Machine Check, INIT and NMI

<table>
<thead>
<tr>
<th>Function</th>
<th>Called after Machine Check, INIT and NMI</th>
</tr>
</thead>
<tbody>
<tr>
<td>GetTime()</td>
<td>Yes, even if previously busy</td>
</tr>
<tr>
<td>GetVariable()</td>
<td>Yes, even if previously busy</td>
</tr>
<tr>
<td>GetNextVariableName()</td>
<td>Yes, even if previously busy</td>
</tr>
<tr>
<td>QueryVariableInfo()</td>
<td>Yes, even if previously busy</td>
</tr>
<tr>
<td>SetVariable()</td>
<td>Yes, even if previously busy</td>
</tr>
<tr>
<td>UpdateCapsule()</td>
<td>Yes, even if previously busy</td>
</tr>
<tr>
<td>QueryCapsuleCapabilities()</td>
<td>Yes, even if previously busy</td>
</tr>
<tr>
<td>ResetSystem()</td>
<td>Yes, even if previously busy</td>
</tr>
</tbody>
</table>
8.2 Variable Services

Variables are defined as key/value pairs that consist of identifying information plus attributes (the key) and arbitrary data (the value). Variables are intended for use as a means to store data that is passed between the EFI environment implemented in the platform and EFI OS loaders and other applications that run in the EFI environment.

Although the implementation of variable storage is not defined in this specification, variables must be persistent in most cases. This implies that the EFI implementation on a platform must arrange it so that variables passed in for storage are retained and available for use each time the system boots, at least until they are explicitly deleted or overwritten. Provision of this type of nonvolatile storage may be very limited on some platforms, so variables should be used sparingly in cases where other means of communicating information cannot be used.

Table 8-3 lists the variable services functions described in this section:

<table>
<thead>
<tr>
<th>Name</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>GetVariable()</td>
<td>Runtime</td>
<td>Returns the value of a variable.</td>
</tr>
<tr>
<td>GetNextVariableName</td>
<td>Runtime</td>
<td>Enumerates the current variable names.</td>
</tr>
<tr>
<td>SetVariable</td>
<td>Runtime</td>
<td>Sets the value of a variable.</td>
</tr>
<tr>
<td>QueryVariableInfo</td>
<td>Runtime</td>
<td>Returns information about the EFI variables</td>
</tr>
</tbody>
</table>

GetVariable()

Summary

Returns the value of a variable.

Prototype

```c
typedef EFI_STATUS GetVariable ( 
    IN CHAR16   *VariableName, 
    IN EFI_GUID *VendorGuid,  
    OUT UINT32 *Attributes OPTIONAL, 
    IN OUT UINTN *DataSize, 
    OUT VOID *Data OPTIONAL  
);
```

Parameters

- **VariableName**: A Null-terminated string that is the name of the vendor's variable.
- **VendorGuid**: A unique identifier for the vendor. Type **EFI_GUID** is defined in the **EFI_BOOT_SERVICES.InstallProtocolInterface()** function description.
Attributes

If not `NULL`, a pointer to the memory location to return the attributes bitmask for the variable. See “Related Definitions.” If not NULL, then Attributes is set on output both when `EFI_SUCCESS` and when `EFI_BUFFER_TOO_SMALL` is returned.

DataSize

On input, the size in bytes of the return Data buffer. On output the size of data returned in Data.

Data

The buffer to return the contents of the variable. May be `NULL` with a zero DataSize in order to determine the size buffer needed.

Related Definitions

```
//*******************************************************
// Variable Attributes
//*******************************************************
#define EFI_VARIABLE_NON_VOLATILE                            0x00000001
#define EFI_VARIABLE_BOOTSERVICE_ACCESS                      0x00000002
#define EFI_VARIABLE_RUNTIME_ACCESS                          0x00000004
#define EFI_VARIABLE_HARDWARE_ERROR_RECORD                   0x00000008 \  //This attribute is identified by the mnemonic 'HR' elsewhere
    //in this specification.
#define EFI_VARIABLE_AUTHENTICATED_WRITE_ACCESS              0x00000010
    //NOTE: EFI_VARIABLE_AUTHENTICATED_WRITE_ACCESS is deprecated
    //and should be considered reserved.
#define EFI_VARIABLE_TIME_BASED_AUTHENTICATED_WRITE_ACCESS \ 0x00000020
#define EFI_VARIABLE_APPEND_WRITE                            0x00000040
#define EFI_VARIABLE_ENHANCED_AUTHENTICATED_ACCESS           0x00000080
    //This attribute indicates that the variable payload begins
    //with an EFI_VARIABLE_AUTHENTICATION_3 structure, and
    //potentially more structures as indicated by fields of this
    //structure. See definition below and in SetVariable().
```

Description

Each vendor may create and manage its own variables without the risk of name conflicts by using a unique `VendorGuid`. When a variable is set its Attributes are supplied to indicate how the data variable should be stored and maintained by the system. The attributes affect when the variable may be accessed and volatility of the data. If `EFI_BOOT_SERVICES.ExitBootServices()` has already been executed, data variables without the `EFI_VARIABLE_RUNTIME_ACCESS` attribute set will not be visible to `GetVariable()` and will return an `EFI_NOT_FOUND` error.

If the Data buffer is too small to hold the contents of the variable, the error `EFI_BUFFER_TOO_SMALL` is returned and `DataSize` is set to the required buffer size to obtain the data.

The `EFI_VARIABLE_TIME_BASED_AUTHENTICATED_WRITE_ACCESS` and the `EFI_VARIABLE_AUTHENTICATED_WRITE_ACCESS` attributes may both be set in the returned Attributes bitmask parameter of a `GetVariable()` call, though it should be noted that the `EFI_VARIABLE_AUTHENTICATED_WRITE_ACCESS` attribute is deprecated and should no longer be used.
**EFI_VARIABLE_APPEND_WRITE** attribute will never be set in the returned Attributes bitmask parameter.

Variables stored with the EFI_VARIABLE_ENHANCED_AUTHENTICATED_ACCESS attribute set will return metadata in addition to variable data when GetVariable() is called. If a GetVariable() call indicates that this attribute is set, the GetVariable() payload must be interpreted according to the metadata headers. In addition to the headers described in SetVariable(), the following header is used to indicate what certificate may be currently associated with a variable.

```c
//
// EFI_VARIABLE_AUTHENTICATION_3_CERT_ID descriptor
// An extensible structure to identify a unique x509 cert
// associated with a given variable
//
#define EFI_VARIABLE_AUTHENTICATION_3_CERT_ID_SHA256 1

typedef struct {
    UINT8         Type;
    UINT32        IdSize;
    // UINT8      Id[IdSize];
    } EFI_VARIABLE_AUTHENTICATION_3_CERT_ID;
```

Type

Identifies the type of ID that is returned and how the ID should be interpreted.

IdSize

Indicates the size of the Id buffer that follows this field in the structure.

Id (Not a formal structure member)

This is a unique identifier for the associated certificate as defined by the Type field. For CERT_ID_SHA256, the buffer will be a SHA-256 digest of the tbsCertificate (To Be Signed Certificate data defined in x509) data for the cert.

When the attribute **EFI_VARIABLE_ENHANCED_AUTHENTICATED_ACCESS** is set, the Data buffer shall be interpreted as follows:

// NOTE: “||“ indicates concatenation.

// Example: **EFI_VARIABLE_AUTHENTICATION_3_TIMESTAMP_TYPE**

```c
EFI_VARIABLE_AUTHENTICATION_3 || EFI_TIME || EFI_VARIABLE_AUTHENTICATION_3_CERT_ID || Data
```
// Example: \texttt{EFI\_VARIABLE\_AUTHENTICATION\_3\_NONCE\_TYPE}
\texttt{EFI\_VARIABLE\_AUTHENTICATION\_3 || EFI\_VARIABLE\_AUTHENTICATION\_3\_NONCE ||
EFI\_VARIABLE\_AUTHENTICATION\_3\_CERT\_ID || Data}

NOTE: The MetadataSize field of the \texttt{EFI\_VARIABLE\_AUTHENTICATION\_3} structure in each of these examples does not include any \texttt{WIN\_CERTIFICATE\_UEFI\_GUID} structures. These structures are used in the \texttt{SetVariable()} interface, not \texttt{GetVariable()}, as described in the above examples.

\textbf{Status Codes Returned}

<table>
<thead>
<tr>
<th>Code</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>EFI_SUCCESS</td>
<td>The function completed successfully.</td>
</tr>
<tr>
<td>EFI_NOT_FOUND</td>
<td>The variable was not found.</td>
</tr>
<tr>
<td>EFI_BUFFER_TOO_SMALL</td>
<td>The $\textit{DataSize}$ is too small for the result. $\textit{DataSize}$ has been updated with the size needed to complete the request. If $\textit{Attributes}$ is not NULL, then the attributes bitmask for the variable has been stored to the memory location pointed-to by $\textit{Attributes}$.</td>
</tr>
<tr>
<td>EFI_INVALID_PARAMETER</td>
<td>$\textit{VariableName}$ is NULL.</td>
</tr>
<tr>
<td>EFI_INVALID_PARAMETER</td>
<td>$\textit{VendorGuid}$ is NULL.</td>
</tr>
<tr>
<td>EFI_INVALID_PARAMETER</td>
<td>$\textit{DataSize}$ is NULL.</td>
</tr>
<tr>
<td>EFI_INVALID_PARAMETER</td>
<td>The $\textit{DataSize}$ is not too small and $\textit{Data}$ is NULL.</td>
</tr>
<tr>
<td>EFI_DEVICE_ERROR</td>
<td>The variable could not be retrieved due to a hardware error.</td>
</tr>
<tr>
<td>EFI_SECURITY_VIOLATION</td>
<td>The variable could not be retrieved due to an authentication failure.</td>
</tr>
<tr>
<td>EFI_UNSUPPORTED</td>
<td>After \texttt{ExitBootServices()} has been called, this return code may be returned if no variable storage is supported. The platform should describe this runtime service as unsupported at runtime via an \texttt{EFI_RT_PROPERTIES_TABLE} configuration table.</td>
</tr>
</tbody>
</table>

\textbf{GetNextVariableName()}

\textbf{Summary}
Enumerates the current variable names.

\textbf{Prototype}
typedef EFI_STATUS
GetNextVariableName (  
IN OUT UINTN *VariableNameSize,  
IN OUT CHAR16 *VariableName,  
IN OUT EFI_GUID *VendorGuid  
);

Parameters

VariableNameSize  The size of the VariableName buffer. The size must be large enough to fit input string supplied in VariableName buffer.

VariableName  On input, supplies the last VariableName that was returned by GetNextVariableName(). On output, returns the Null-terminated string of the current variable.

VendorGuid  On input, supplies the last VendorGuid that was returned by GetNextVariableName(). On output, returns the VendorGuid of the current variable. Type EFI_GUID is defined in the EFI_BOOT_SERVICES.InstallProtocolInterface() function description.

Description

GetNextVariableName() is called multiple times to retrieve the VariableName and VendorGuid of all variables currently available in the system. On each call to GetNextVariableName() the previous results are passed into the interface, and on output the interface returns the next variable name data. When the entire variable list has been returned, the error EFI_NOT_FOUND is returned.

Note that if EFI_BUFFER_TOO_SMALL is returned, the VariableName buffer was too small for the next variable. When such an error occurs, the VariableNameSize is updated to reflect the size of buffer needed. In all cases when calling GetNextVariableName() the VariableNameSize must not exceed the actual buffer size that was allocated for VariableName. The VariableNameSize must not be smaller the size of the variable name string passed to GetNextVariableName() on input in the VariableName buffer.

To start the search, a Null-terminated string is passed in VariableName; that is, VariableName is a pointer to a Null character. This is always done on the initial call to GetNextVariableName(). When VariableName is a pointer to a Null character, VendorGuid is ignored. GetNextVariableName() cannot be used as a filter to return variable names with a specific GUID. Instead, the entire list of variables must be retrieved, and the caller may act as a filter if it chooses. Calls to SetVariable() between calls to GetNextVariableName() may produce unpredictable results. If a VariableName buffer on input is not a Null-terminated string, EFI_INVALID_PARAMETER is returned. If input values of VariableName and VendorGuid are not a name and GUID of an existing variable, EFI_INVALID_PARAMETER is returned.

Once EFI_BOOT_SERVICES.ExitBootServices() is performed, variables that are only visible during boot service will no longer be returned. To obtain the data contents or attribute for a variable returned by GetNextVariableName(), the GetVariable() interface is used.
Status Codes Returned

<table>
<thead>
<tr>
<th>Code</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>EFI_SUCCESS</td>
<td>The function completed successfully.</td>
</tr>
<tr>
<td>EFI_NOT_FOUND</td>
<td>The next variable was not found.</td>
</tr>
<tr>
<td>EFI_BUFFER_TOO_SMALL</td>
<td>The VariableNameSize is too small for the result. VariableNameSize has been updated with the size needed to complete the request.</td>
</tr>
<tr>
<td>EFI_INVALID_PARAMETER</td>
<td>VariableNameSize is NULL.</td>
</tr>
<tr>
<td>EFI_INVALID_PARAMETER</td>
<td>VariableName is NULL.</td>
</tr>
<tr>
<td>EFI_INVALID_PARAMETER</td>
<td>VendorGuid is NULL.</td>
</tr>
<tr>
<td>EFI_INVALID_PARAMETER</td>
<td>The input values of VariableName and VendorGuid are not a name and GUID of an existing variable.</td>
</tr>
<tr>
<td>EFI_INVALID_PARAMETER</td>
<td>Null-terminator is not found in the first VariableNameSize bytes of the input VariableName buffer.</td>
</tr>
<tr>
<td>EFI_DEVICE_ERROR</td>
<td>The variable name could not be retrieved due to a hardware error.</td>
</tr>
<tr>
<td>EFI_UNSUPPORTED</td>
<td>After ExitBootServices() has been called, this return code may be returned if no variable storage is supported. The platform should describe this runtime service as unsupported at runtime via an EFI_RT_PROPERTIES_TABLE configuration table.</td>
</tr>
</tbody>
</table>

SetVariable()

Summary
Sets the value of a variable. This service can be used to create a new variable, modify the value of an existing variable, or to delete an existing variable.

Prototype

```c
typedef
  EFI_STATUS
  SetVariable ( 
    IN CHAR16   *VariableName,
    IN EFI_GUID *VendorGuid, 
    IN UINT32   Attributes, 
    IN UINTN    DataSize,    
    IN VOID     *Data        
  );
```

Parameters

- **VariableName**: A Null-terminated string that is the name of the vendor's variable. Each VariableName is unique for each VendorGuid. VariableName must contain 1 or more characters. If VariableName is an empty string, then EFI_INVALID_PARAMETER is returned.
- **VendorGuid**: A unique identifier for the vendor. Type EFI_GUID is defined in the EFI_BOOT_SERVICES.InstallProtocolInterface() function description.
Attributes

Attributes bitmask to set for the variable. Refer to the GetVariable() function description.

DataSize

The size in bytes of the Data buffer. Unless the EFI_VARIABLE_APPEND_WRITE, EFI_VARIABLE_AUTHENTICATED_WRITE_ACCESS, EFI_VARIABLE_ENHANCED_AUTHENTICATED_ACCESS, or EFI_VARIABLE_TIME_BASED_AUTHENTICATED_WRITE_ACCESS attribute is set, a size of zero causes the variable to be deleted. When the EFI_VARIABLE_APPEND_WRITE attribute is set, then a SetVariable() call with a DataSize of zero will not cause any change to the variable value (the timestamp associated with the variable may be updated however, even if no new data value is provided; see the description of the EFI_VARIABLE_AUTHENTICATION_2 descriptor below). In this case the DataSize will not be zero since the EFI_VARIABLE_AUTHENTICATION_2 descriptor will be populated.

Data

The contents for the variable.

Related Definitions

//*******************************************************
// Variable Attributes
//*******************************************************

// NOTE: This interface is deprecated and should no longer be used!

// EFI_VARIABLE_AUTHENTICATION descriptor

// A counter-based authentication method descriptor template

typedef struct {
    UINT64 MonotonicCount;
    WIN_CERTIFICATE_UEFI_GUID AuthInfo;
} EFI_VARIABLE_AUTHENTICATION;

MonotonicCount

Included in the signature of AuthInfo. Used to ensure freshness/no replay. Incremented during each "Write" access.

AuthInfo

Provides the authorization for the variable access. It is a signature across the variable data and the Monotonic Count value. Caller uses Private key that is associated with a public key that has been provisioned via the key exchange.
// EFI_VARIABLE_AUTHENTICATION_2 descriptor
// A time-based authentication method descriptor template
typedef struct {
    EFI_TIME TimeStamp;
    WIN_CERTIFICATE_UEFI_GUID AuthInfo;
} EFI_VARIABLE_AUTHENTICATION_2;

TimeStamp
Time associated with the authentication descriptor. For the TimeStamp value, components Pad1, Nanosecond, TimeZone, Daylight and Pad2 shall be set to 0. This means that the time shall always be expressed in GMT.

AuthInfo
Provides the authorization for the variable access. Only a CertType of EFI_CERT_TYPE_PKCS7_GUID is accepted.

// EFI_VARIABLE_AUTHENTICATION_3 descriptor
// An extensible implementation of the Variable Authentication structure.
#define EFI_VARIABLE_AUTHENTICATION_3_TIMESTAMP_TYPE 1
#define EFI_VARIABLE_AUTHENTICATION_3_NONCE_TYPE 2
typedef struct {
    UINT8 Version;
    UINT8 Type;
    UINT32 MetadataSize;
    UINT32 Flags;
} EFI_VARIABLE_AUTHENTICATION_3;

Version
This field is used in case the EFI_VARIABLE_AUTHENTICATION_3 structure itself ever requires updating. For now, it is hardcoded to “0x1”.

Type
Declares what structure immediately follows this structure in the Variable Data payload. For EFI_VARIABLE_AUTHENTICATION_3_TIMESTAMP_TYPE, it will be an instance of EFI_TIME (for the TimeStamp). For EFI_VARIABLE_AUTHENTICATION_3_NONCE_TYPE the structure will be an instance of EFI_VARIABLE_AUTHENTICATION_3_NONCE. This structure is defined below. Note that none of these structures contains a
WIN_CERTIFICATE_UEFI_GUID structure. See Section 8.2.1 for an explanation of structure sequencing.

**MetadataSize**

Declares the size of all variable authentication metadata (data related to the authentication of the variable that is not variable data itself), including this header structure, and type-specific structures (e.g., EFI_VARIABLE_AUTHENTICATION_3_NONCE), and any WIN_CERTIFICATE_UEFI_GUID structures.

**Flags**

Bitfield indicating any optional configuration for this call. Currently, the only defined value is:

```c
#define EFI_VARIABLE_ENHANCED_AUTH_FLAG_UPDATE_CERT 0x00000001
```

The presence of this flag on SetVariable() indicates that there are two instances of the WIN_CERTIFICATE_UEFI_GUID structure following the type-specific structures. The first instance describes the new cert to be set as the authority for the variable. The second is the signed data to authorize the current updated.

NOTE: All other bits are currently Reserved on SetVariable().

NOTE: All flags are reserved on GetVariable().

```c
//
// EFI_VARIABLE_AUTHENTICATION_3_NONCE descriptor
//
// A nonce-based authentication method descriptor template. This structure will always be followed by a
// WIN_CERTIFICATE_UEFI_GUID structure.
//
typedef struct {
    UINT32           NonceSize;
    // UINT8         Nonce[NonceSize];
} EFI_VARIABLE_AUTHENTICATION_3_NONCE;
```

**NonceSize**

Indicates the size of the Nonce buffer that follows this field in the structure. Must not be 0.

**Nonce (Not a formal structure member)**

Unique, random value that guarantees a signed payload cannot be shared between multiple machines or machine families. On SetVariable(), if the Nonce field is all 0’s, the host machine will try to use an internally generated random number. Will return EFI_UNSUPPORTED if not possible. Also, on SetVariable() if the variable already exists and the nonce is identical to the current nonce, will return EFI_INVALID_PARAMETER.
Description

Variables are stored by the firmware and may maintain their values across power cycles. Each vendor may create and manage its own variables without the risk of name conflicts by using a unique VendorGuid.

Each variable has Attributes that define how the firmware stores and maintains the data value. If the EFI_VARIABLE_NON_VOLATILE attribute is not set, the firmware stores the variable in normal memory and it is not maintained across a power cycle. Such variables are used to pass information from one component to another. An example of this is the firmware’s language code support variable. It is created at firmware initialization time for access by EFI components that may need the information, but does not need to be backed up to nonvolatile storage.

EFI_VARIABLE_NON_VOLATILE variables are stored in fixed hardware that has a limited storage capacity; sometimes a severely limited capacity. Software should only use a nonvolatile variable when absolutely necessary. In addition, if software uses a nonvolatile variable it should use a variable that is only accessible at boot services time if possible.

A variable must contain one or more bytes of Data. Unless the EFI_VARIABLE_APPEND_WRITE, EFI_VARIABLE_TIME_BASED_AUTHENTICATED_WRITE_ACCESS, or EFI_VARIABLE_ENHANCED_AUTHENTICATED_ACCESS attribute is set (see below), using SetVariable() with a DataSize of zero will cause the entire variable to be deleted. The space consumed by the deleted variable may not be available until the next power cycle.

If a variable with matching name, GUID, and attributes already exists, its value is updated.

The Attributes have the following usage rules:

- If a preexisting variable is rewritten with different attributes, SetVariable() shall not modify the variable and shall return EFI_INVALID_PARAMETER. The only exception to this is when the only attribute differing is EFI_VARIABLE_APPEND_WRITE. In such cases the call’s successful outcome or not is determined by the actual value being written. There are two exceptions to this rule:
  - If a preexisting variable is rewritten with no access attributes specified, the variable will be deleted.
  - EFI_VARIABLE_APPEND_WRITE attribute presents a special case. It is acceptable to rewrite the variable with or without EFI_VARIABLE_APPEND_WRITE attribute.

- Setting a data variable with no access attributes causes it to be deleted.

- EFI_VARIABLE_AUTHENTICATED_WRITE_ACCESS is deprecated and should not be used. Platforms should return EFI_UNSUPPORTED if a caller to SetVariable() specifies this attribute.

- Unless the EFI_VARIABLE_APPEND_WRITE, EFI_VARIABLE_TIME_BASED_AUTHENTICATED_WRITE_ACCESS, or EFI_VARIABLE_ENHANCED_AUTHENTICATED_WRITE_ACCESS attribute is set, setting a data variable with zero DataSize specified, causes it to be deleted.

- Runtime access to a data variable implies boot service access. Attributes that have EFI_VARIABLE_RUNTIME_ACCESS set must also have EFI_VARIABLE_BOOTSERVICE_ACCESS set. The caller is responsible for following this rule.
• Once EFI_BOOT_SERVICES.ExitBootServices() is performed, data variables that did not have EFI_VARIABLE_RUNTIME_ACCESS set are no longer visible to GetVariable().

• Once ExitBootServices() is performed, only variables that have EFI_VARIABLE_RUNTIME_ACCESS and EFI_VARIABLE_NON_VOLATILE set can be set with SetVariable(). Variables that have runtime access but that are not nonvolatile are read-only data variables once ExitBootServices() is performed.

When the EFI_VARIABLE_ENHANCED_AUTHENTICATED_ACCESS attribute is set in a SetVariable() call, the authentication shall use the EFI_VARIABLE_AUTHENTICATION_3 descriptor, which will be followed by any descriptors indicated in the Type and Flags fields.

• When the EFI_VARIABLE_TIME_BASED_AUTHENTICATED_WRITE_ACCESS attribute is set in a SetVariable() call, the authentication shall use the EFI_VARIABLE_AUTHENTICATION_2 descriptor.

• If both the EFI_VARIABLE_TIME_BASED_AUTHENTICATED_WRITE_ACCESS and the EFI_VARIABLE_ENHANCED_AUTHENTICATED_ACCESS attribute are set in a SetVariable() call, then the firmware must return EFI_INVALID_PARAMETER.

• If the EFI_VARIABLE_APPEND_WRITE attribute is set in a SetVariable() call, then any existing variable value shall be appended with the value of the Data parameter. If the firmware does not support the append operation, then the SetVariable() call shall return EFI_INVALID_PARAMETER.

• If the EFI_VARIABLE_TIME_BASED_AUTHENTICATED_WRITE_ACCESS attribute is set in a SetVariable() call, and firmware does not support signature type of the certificate included in the EFI_VARIABLE_AUTHENTICATION_2 descriptor, then the SetVariable() call shall return EFI_INVALID_PARAMETER. The list of signature types supported by the firmware is defined by the SignatureSupport variable. Signature type of the certificate is defined by its digest and encryption algorithms.

• If the EFI_VARIABLE_HARDWARE_ERROR_RECORD attribute is set, VariableName and VendorGuid must comply with the rules stated in Section 8.2.4.2 and Appendix P. Otherwise, the SetVariable() call shall return EFI_INVALID_PARAMETER.

• Globally Defined Variables must be created with the attributes defined in Table 3-1 of the Boot Manager chapter. If a globally defined variable is created with the wrong attributes, the result is indeterminate and may vary between implementations.

• If using the EFI_VARIABLE_ENHANCED_AUTHENTICATED_ACCESS interface to update the cert authority for a given variable, it is valid for the Data region of the payload to be empty. This would update the cert without modifying the data itself. If the Data region is empty AND no NewCert is specified, the variable will be deleted (assuming all authorizations are verified).

• Secure Boot Policy Variable must be created with the EFI_VARIABLE_TIME_BASED_AUTHENTICATED_WRITE_ACCESS attribute set, and the authentication shall use the EFI_VARIABLE_AUTHENTICATION_2 descriptor. If the appropriate attribute bit is not set, then the firmware shall return EFI_INVALID_PARAMETER.

The only rules the firmware must implement when saving a nonvolatile variable is that it has actually been saved to nonvolatile storage before returning EFI_SUCCESS, and that a partial save is not performed. If power fails during a call to SetVariable() the variable may contain its previous value, or its new value. In addition there is no read, write, or delete security protection.
To delete a variable created with the `EFI_VARIABLE_TIME_BASED_AUTHENTICATED_WRITE_ACCESS` attribute, `SetVariable` must be used with attributes matching the existing variable and the `DataSize` set to the size of the `AuthInfo` descriptor. The `Data` buffer must contain an instance of the `AuthInfo` descriptor which will be validated according to the steps in the appropriate section above referring to updates of Authenticated variables. An attempt to delete a variable created with the `EFI_VARIABLE_TIME_BASED_AUTHENTICATED_WRITE_ACCESS` attribute for which the prescribed `AuthInfo` validation fails or when called using `DataSize` of zero will fail with an `EFI_SECURITY_VIOLATION` status.

To delete a variable created with the `EFI_VARIABLE_ENHANCED_AUTHENTICATED_ACCESS` attribute, `SetVariable` must be used with attributes matching the existing variable and the `DataSize` set to the size of the entire payload including all descriptors and certificates. The `Data` buffer must contain an instance of the `EFI_VARIABLE_AUTHENTICATION_3` descriptor which will indicate how to validate the payload according to the description in Section 8.2.1. An attempt to delete a variable created with the `EFI_VARIABLE_ENHANCED_AUTHENTICATED_ACCESS` attribute for which the prescribed validation fails or when called using `DataSize` of zero will fail with an `EFI_SECURITY_VIOLATION` status.

**Status Codes Returned**

<table>
<thead>
<tr>
<th>Status Code</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>EFI_SUCCESS</td>
<td>The firmware has successfully stored the variable and its data as defined by the Attributes.</td>
</tr>
<tr>
<td>EFI_INVALID_PARAMETER</td>
<td>An invalid combination of attribute bits, name, and GUID was supplied, or the <code>DataSize</code> exceeds the maximum allowed.</td>
</tr>
<tr>
<td>EFI_INVALID_PARAMETER</td>
<td><code>VariableName</code> is an empty string.</td>
</tr>
<tr>
<td>EFI_OUT_OF_RESOURCES</td>
<td>Not enough storage is available to hold the variable and its data.</td>
</tr>
<tr>
<td>EFI_DEVICE_ERROR</td>
<td>The variable could not be saved due to a hardware failure.</td>
</tr>
<tr>
<td>EFI_WRITE_PROTECTED</td>
<td>The variable in question is read-only.</td>
</tr>
<tr>
<td>EFI_WRITE_PROTECTED</td>
<td>The variable in question cannot be deleted.</td>
</tr>
<tr>
<td>EFI_SECURITY_VIOLATION</td>
<td>The variable could not be written due to <code>EFI_VARIABLE_ENHANCED_AUTHENTICATED_ACCESS</code> or <code>EFI_VARIABLE_TIME_BASED_AUTHENTICATED_WRITE_ACCESS</code> being set, but the payload does NOT pass the validation check carried out by the firmware.</td>
</tr>
<tr>
<td>EFI_NOT_FOUND</td>
<td>The variable trying to be updated or deleted was not found.</td>
</tr>
<tr>
<td>EFI_UNSUPPORTED</td>
<td>This call is not supported by this platform at the time the call is made. The platform should describe this runtime service as unsupported at runtime via an <code>EFI_RT_PROPERTIES_TABLE</code> configuration table.</td>
</tr>
</tbody>
</table>

**QueryVariableInfo()**

**Summary**

Returns information about the EFI variables.
Prototype

```c
typedef EFI_STATUS
QueryVariableInfo (
    IN UINT32 Attributes,
    OUT UINT64 *MaximumVariableStorageSize,
    OUT UINT64 *RemainingVariableStorageSize,
    OUT UINT64 *MaximumVariableSize
);
```

**Attributes**
Attributes bitmask to specify the type of variables on which to return information. Refer to the `GetVariable()` function description. The `EFI_VARIABLE_APPEND_WRITE` attribute, if set in the attributes bitmask, will be ignored.

**MaximumVariableStorageSize**
On output the maximum size of the storage space available for the EFI variables associated with the attributes specified.

**RemainingVariableStorageSize**
Returns the remaining size of the storage space available for EFI variables associated with the attributes specified.

**MaximumVariableSize**
Returns the maximum size of an individual EFI variable associated with the attributes specified.

**Description**
The `QueryVariableInfo()` function allows a caller to obtain the information about the maximum size of the storage space available for the EFI variables, the remaining size of the storage space available for the EFI variables and the maximum size of each individual EFI variable, associated with the attributes specified.

The `MaximumVariableSize` value will reflect the overhead associated with the saving of a single EFI variable with the exception of the overhead associated with the length of the string name of the EFI variable.

The returned `MaximumVariableStorageSize, RemainingVariableStorageSize, MaximumVariableSize` information may change immediately after the call based on other runtime activities including asynchronous error events. Also, these values associated with different attributes are not additive in nature.

After the system has transitioned into runtime (after `ExitBootServices()` is called), an implementation may not be able to accurately return information about the Boot Services variable store. In such cases, `EFI_INVALID_PARAMETER` should be returned.
8.2.1 Using the EFI_VARIABLE_AUTHENTICATION_3 descriptor

When the attribute **EFI_VARIABLE_ENHANCED_AUTHENTICATED_ACCESS** is set, the payload buffer (passed into SetVariable() as “Data”) shall be constructed as follows:

// NOTE: “||” indicates concatenation.
// NOTE: “[]” indicates an optional element.

// Example: **EFI_VARIABLE_AUTHENTICATION_3_TIMESTAMP_TYPE**

```plaintext
EFI_VARIABLE_AUTHENTICATION_3 || EFI_TIME || [ NewCert ] || SigningCert || Data
```

// Example: **EFI_VARIABLE_AUTHENTICATION_3_NONCE_TYPE**

```plaintext
EFI_VARIABLE_AUTHENTICATION_3 || EFI_VARIABLE_AUTHENTICATION_3_NONCE || [ NewCert ] || SigningCert || Data
```

In this example, NewCert and SigningCert are both instances of **WIN_CERTIFICATE_UEFI_GUID**. The presence of NewCert is indicated by the **EFI_VARIABLE_AUTHENTICATION_3.Flags** field (see Definition in SetVariable()). If provided – and assuming the payload passes all integrity and security verifications – this cert will be set as the new authority for the underlying variable, even if the variable is being newly created.

The NewCert element must have a CertType of **EFI_CERT_TYPE_PKCS7_GUID**, and the CertData must be a DER-encoded SignedData structure per PKCS#7 version 1.5 (RFC 2315), which shall be supported both with and without a DER-encoded ContentInfo structure per PKCS#7 version 1.5. When creating the SignedData structure, the following steps shall be followed:

1. Create a **WIN_CERTIFICATE_UEFI_GUID** structure where CertType is set to **EFI_CERT_TYPE_PKCS7_GUID**.
2. Use the x509 cert being added as the new authority to sign its own tbsCertificate data.
3. Construct a DER-encoded PKCS #7 version 1.5 SignedData (see [RFC2315]) with the signed content as follows:
   a) SignedData.version shall be set to 1.
   b) SignedData.digestAlgorithms shall contain the digest algorithm used when preparing the signature. Only a digest algorithm of SHA-256 is accepted.
c SignedData.contentInfo.contentType shall be set to id-data.
d SignedData.contentInfo.content shall be the tbsCertificate data that was signed for the
new x509 cert.
e SignedData.certificates shall contain, at a minimum, the signer’s DER-encoded X.509
certificate.
f SignedData.crls is optional.
g SignedData.signerInfos shall be constructed as:
• SignerInfo.version shall be set to 1.
• SignerInfo.issuerAndSerial shall be present and as in the signer’s certificate.
• SignerInfo.authenticatedAttributes shall not be present.
• SignerInfo.digestEncryptionAlgorithm shall be set to the algorithm used to sign the
data. Only a digest encryption algorithm of RSA with PKCS #1 v1.5 padding
(RSASSA_PKCS1v1_5) is accepted.
• SignerInfo.encryptedDigest shall be present.
• SignerInfo.unauthenticatedAttributes shall not be present.

4. Set the CertData field to the DER-encoded PKCS#7 SignedData value.

A caller to SetVariable() attempting to create, update, or delete a variable with the
EFI_VARIABLE_ENHANCED_AUTHENTICATED_ACCESS set shall perform the following steps to create
the SignedData structure for SigningCert:

1. Create an EFI_VARIABLE_AUTHENTICATION_3 Primary Descriptor with the following values:
   a Version shall be set appropriate to the version of metadata headers being used (currently
       1).
   b Type should be set based on caller specifications (see
       EFI_VARIABLE_AUTHENTICATION_3 descriptor
       under SetVariable()).
   c MetadataSize can be ignored for now, and will be updated when constructing the final
       payload.
   d Flags shall be set based on caller specifications.
2. A Secondary Descriptor may need to be created based on the Type.
   a For EFI_VARIABLE_AUTHENTICATION_3_TIMESTAMP_TYPE type, this will be an
      instance of EFI_TIME set to the current time.
   b For EFI_VARIABLE_AUTHENTICATION_3_NONCE_TYPE type, this will be an instance of
      EFI_VARIABLE_AUTHENTICATION_3_NONCE updated with NonceSize set based on
      caller specifications (must not be zero), and Nonce (informal structure member) set to:
         • All zeros to request that the platform create a random nonce.
         • Caller specified value for a pre-generated nonce.
3. Hash a serialization of the payload. Serialization shall contain the following elements in this
   order:
   a VariableName, VendorGuid, Attributes, and the Secondary Descriptor if it exists for this
      Type.
b Variable’s new value (ie. the Data parameter’s new variable content).

c If this is an update to or deletion of a variable with type
**EFI_VARIABLE_AUTHENTICATION_3_NONCE**, serialize the current nonce. The current
nonce is the one currently associated with this variable, not the one in the Secondary
Descriptor. Serialize only the nonce buffer contents, not the size or any additional data. If
this is an attempt to create a new variable (ie. there is no current nonce), skip this step.

d If the authority cert for this variable is being updated and the
**EFI_VARIABLE_AUTHENTICATION_3.Flags** field indicates the presence of a NewCert
structure, serialize the entire NewCert structure (described at the beginning of this
section).

4. Sign the resulting digest.

5. Create a **WIN_CERTIFICATE_UEFI_GUID** structure where CertType is set to
**EFI_CERT_TYPE_PKCS7_GUID**.

6. Construct a DER-encoded PKCS #7 version 1.5 SignedData (see [RFC2315]) following the steps
described for NewCert (step 3), above, with the following exception:

a SignedData.contentInfo.content shall be absent (the content is provided in the Data
parameter to the SetVariable() call)

7. Construct the final payload for SetVariable() according to the descriptions for “payload buffer”
at the beginning of this section.

8. Update the **EFI_VARIABLE_AUTHENTICATION_3.MetadataSize** field to include all parts
of the final payload except “Data”.

Firmware that implements the SetVariable() services and supports the
**EFI_VARIABLE_ENHANCED_AUTHENTICATED_ACCESS** attribute shall do the following in response to
being called:

1. Read the **EFI_VARIABLE_AUTHENTICATION_3** descriptor to determine what type of
authentication is being performed and how to parse the rest of the payload.

2. Verify that SigningCert.CertType is **EFI_CERT_TYPE_PKCS7_GUID**.

a If **EFI_VARIABLE_AUTHENTICATION_3.Flags** field indicates presence of a NewCert,
verify that NewCert.CertType is **EFI_CERT_TYPE_PKCS7_GUID**.

b If either fails, return **EFI_INVALID_PARAMETER**.

3. If the variable already exists, verify that the incoming type matches the existing type.

4. Verify that any **EFI_TIME** structures have Pad1, Nanosecond, TimeZone, Daylight, and Pad2
fields set to zero.

5. If **EFI_VARIABLE_AUTHENTICATION_3_NONCE_TYPE**:

a Verify that NonceSize is greater than zero. If zero, return **EFI_INVALID_PARAMETER**.

b If incoming nonce is all zeros, confirm that platform supports generating random nonce. If
unsupported, return **EFI_UNSUPPORTED**.

c If nonce is specified and variable already exists, verify that incoming nonce does not match
existing nonce. If identical, return **EFI_INVALID_PARAMETER**.

6. If **EFI_VARIABLE_AUTHENTICATION_3_TIMESTAMP_TYPE** and variable already exists,
verify that new timestamp is chronologically greater than current timestamp.
7. Verify the payload signature by:
   a. Parsing entire payload according to descriptors.
   b. Using descriptor contents (and, if necessary, metadata from existing variable) to construct the serialization described previously in this section (step 3 of the SetVariable() instructions).
   c. Compute the digest and compare with the result of applying the SigningCert’s public key to the signature.

8. If the variable already exists, verify that the SigningCert authority is the same as the authority already associated with the variable.

9. If NewCert is provided, verify the NewCert signature by:
   a. Parsing entire payload according to descriptors.
   b. Compute a digest of the tbsCertificate of x509 certificate in NewCert and compare with the result of applying NewCert’s public key to the signature.
   c. If this fails, return EFI_SECURITY_VIOLATION.

8.2.2 Using the EFI_VARIABLE_AUTHENTICATION_2 descriptor

When the attribute EFI_VARIABLE_TIME_BASED_AUTHENTICATED_WRITE_ACCESS is set, then the Data buffer shall begin with an instance of a complete (and serialized) EFI_VARIABLE_AUTHENTICATION_2 descriptor. The descriptor shall be followed by the new variable value and DataSize shall reflect the combined size of the descriptor and the new variable value. The authentication descriptor is not part of the variable data and is not returned by subsequent calls to GetVariable().

A caller that invokes the SetVariable() service with the EFI_VARIABLE_TIME_BASED_AUTHENTICATED_WRITE_ACCESS attribute set shall do the following prior to invoking the service:

1. Create a descriptor
   Create an EFI_VARIABLE_AUTHENTICATION_2 descriptor where:
   • TimeStamp is set to the current time.
   • AuthInfo.CertType is set to EFI_CERT_TYPE_PKCS7_GUID.

   Note: In certain environments a reliable time source may not be available. In this case, an implementation may still add values to an authenticated variable since the EFI_VARIABLE_APPEND_WRITE attribute, when set, disables timestamp verification (see below). In these instances, the special time value where every component of the EFI_TIME struct including the Day and Month is set to 0 shall be used.

2. Hash the serialization of the values of the VariableName, VendorGuid and Attributes parameters of the SetVariable() call and the TimeStamp component of the EFI_VARIABLE_AUTHENTICATION_2 descriptor followed by the variable’s new value (i.e. the Data parameter’s new variable content). That is, digest = hash (VariableName, VendorGuid, Attributes, TimeStamp, DataNew_variable_content). The NULL character terminating the VariableName value shall not be included in the hash computation.
3. Sign the resulting digest using a selected signature scheme (e.g. PKCS #1 v1.5)

4. Construct a DER-encoded SignedData structure per PKCS#7 version 1.5 (RFC 2315), which shall be supported both with and without a DER-encoded ContentInfo structure per PKCS#7 version 1.5, with the signed content as follows:
   a. SignedData.version shall be set to 1
   b. SignedData.digestAlgorithms shall contain the digest algorithm used when preparing the signature. Only a digest algorithm of SHA-256 is accepted.
   c. SignedData.contentInfo.contentType shall be set to id-data
   d. SignedData.contentInfo.content shall be absent (the content is provided in the Data parameter to the SetVariable() call)
   e. SignedData.certificates shall contain, at a minimum, the signer’s DER-encoded X.509 certificate
   f. SignedData.crls is optional.
   g. SignedData.signerInfos shall be constructed as:
      — SignerInfo.version shall be set to 1
      — SignerInfo.issuerAndSerial shall be present and as in the signer’s certificate — SignerInfo.authenticatedAttributes shall not be present.
      — SignerInfo.digestEncryptionAlgorithm shall be set to the algorithm used to sign the data. Only a digest encryption algorithm of RSA with PKCS #1 v1.5 padding (RSASSA_PKCS1v1_5) is accepted.
      — SignerInfo.encryptedDigest shall be present
      — SignerInfo.unauthenticatedAttributes shall not be present.

5. Set AuthInfo.CertData to the DER-encoded PKCS #7 SignedData value.

6. Construct Data parameter: Construct the SetVariable()’s Data parameter by concatenating the complete, serialized EFI_VARIABLE_AUTHENTICATION_2 descriptor with the new value of the variable (DataNew_variable_content).

Firmware that implements the SetVariable() service and supports the EFI_VARIABLE_TIME_BASED_AUTHENTICATED_WRITE_ACCESS attribute shall do the following in response to being called:

1. Verify that the correct AuthInfo.CertType (EFI_CERT_TYPE_PKCS7_GUID) has been used and that the AuthInfo.CertData value parses correctly as a PKCS #7 SignedData value
2. Verify that Pad1, Nanosecond, TimeZone, Daylight and Pad2 components of the TimeStamp value are set to zero. Unless the EFI_VARIABLE_APPEND_WRITE attribute is set, verify that the TimeStamp value is later than the current timestamp value associated with the variable.
3. If the variable SetupMode==1, and the variable is a secure boot policy variable, then the firmware implementation shall consider the checks in the following steps 4 and 5 to have passed, and proceed with updating the variable value as outlined below.
4. Verify the signature by:
   — extracting the EFI_VARIABLE_AUTHENTICATION_2 descriptor from the Data buffer;
using the descriptor contents and other parameters to (a) construct the input to the
digest algorithm; (b) computing the digest; and (c) comparing the digest with the result of
applying the signer’s public key to the signature.

5. If the variable is the global PK variable or the global KEK variable, verify that the signature has
been made with the current Platform Key.
   • If the variable is the “db”, “dbt”, “dbr”, or “dbx” variable mentioned in step 3, verify that
     the signer’s certificate chains to a certificate in the Key Exchange Key database (or that the
     signature was made with the current Platform Key).
   • If the variable is the "OsRecoveryOrder" or "OsRecovery####" variable mentioned in step 3, verify that
     the signer’s certificate chains to a certificate in the "dbr" database or the Key
     Exchange Key database, or that the signature was made with the current Platform Key.
   • Otherwise, if the variable is none of the above, it shall be designated a Private
     Authenticated Variable. If the Private Authenticated Variable does not exist, then the CN of
     the signing certificate’s Subject and the hash of the tbsCertificate of the top-level issuer
     certificate (or the signing certificate itself if no other certificates are present or the
     certificate chain is of length 1) in SignedData.certificates is registered for use in subsequent
     verifications of this variable. Implementations may store just a single hash of these two
     elements to reduce storage requirements. If the Private Authenticated variable previously
     existed, that the signer’s certificate chains to the information previously associated with
     the variable. Observe that because no revocation list exists for them, if any member of the
     certificate chain is compromised, the only method to revoke trust in a certificate for a
     Private Authenticated Variable is to delete the variable, re-issue all certificate authorities in
     the chain, and re-create the variable using the new certificate chain. As such, the remaining
     benefits may be strong identification of the originator, or compliance with some certificate
     authority policy. Further note that the PKCS7 bundle for the authenticated variable update
     must contain the signing certificate chain, through and including the full certificate of the
     desired trust anchor. The trust anchor might be a mid-level certificate or root, though
     many roots may be unsuitable trust anchors due to the number of CAs they issue for
     different purposes. Some tools require non-default parameters to include the trust anchor
     certificate.

The driver shall update the value of the variable only if all of these checks pass. If any of the checks fails,
firmware must return EFI_SECURITY_VIOLATION.

The firmware shall perform an append to an existing variable value only if the
EFI_VARIABLE_APPEND_WRITE attribute is set.

For variables with the GUID EFI_IMAGE_SECURITY_DATABASE_GUID (i.e. where the data buffer is
formatted as EFI_SIGNATURE_LIST), the driver shall not perform an append of
EFI_SIGNATURE_DATA values that are already part of the existing variable value.

Note: This situation is not considered an error, and shall in itself not cause a status code other than
EFI_SUCCESS to be returned or the timestamp associated with the variable not to be updated.

The firmware shall associate the new timestamp with the updated value (in the case when the
EFI_VARIABLE_APPEND_WRITE attribute is set, this only applies if the new TimeStamp value is later
than the current timestamp associated with the variable).

If the variable did not previously exist, and is not one of the variables listed in step 3 above, then
firmware shall associate the signer’s public key with the variable for future verification purposes.
8.2.3 Using the EFI_VARIABLE_AUTHENTICATION descriptor

**Note:** This interface is deprecated and should no longer be used! It will be removed from future versions of the spec.

When the attribute `EFI_VARIABLE_AUTHENTICATED_WRITE_ACCESS` is set, but the `EFI_VARIABLE_TIME_BASED_AUTHENTICATED_WRITE_ACCESS` is not set (i.e. when the `EFI_VARIABLE_AUTHENTICATION` descriptor is used), then the `Data` buffer shall begin with an instance of the authentication descriptor `AuthInfo` prior to the data payload and `DataSize` should reflect the data and descriptor size. The authentication descriptor is not part of the variable data and is not returned by the subsequent calls to `GetVariable`. The caller shall digest the Monotonic Count value and the associated data for the variable update using the SHA-256 1-way hash algorithm. The ensuing the 32-byte digest will be signed using the private key associated w/ the public 2048-bit RSA key `PublicKey` described in the `EFI_CERT_BLOCK_RSA_2048_SHA256` structure.

The `WIN_CERTIFICATE` shall be used to describe the signature of the Variable data `*Data`. In addition, the signature will also include the `MonotonicCount` value to guard against replay attacks. The `MonotonicCount` value must be increased by the caller prior to an update of the `*Data` when the `EFI_VARIABLE_AUTHENTICATED_WRITE_ACCESS` is set.

From the `EFI_CERT_BLOCK_RSA_2048_SHA256`, the `HashType` will be `EFI_SHA256_HASH` and the `ANYSIZE_ARRAY` of `Signature` will be 256. The `WIN_CERTIFICATE_PKCS1_15` could have been used but was not for the following reason: There are possibly various different principals to create authenticated variables, so the public key corresponding to a given principal is added to the `EFI_CERT_BLOCK_RSA_2048_SHA256` within the `WIN_CERTIFICATE`. This does not lend cryptographic value so much as it provides something akin to a handle for the platform firmware to use during its verification operation.

The `MonotonicCount` value must be strictly greater for each successive variable update operation. This allows for ensuring freshness of the update operation and defense against replay attacks (i.e., if someone had the value of a former `AuthInfo`, such as a Man-in-the-Middle they could not re-invogue that same update session). For maintenance, the party who initially provisioned the variable (i.e., caller of `SetVariable`) and set the monotonic count will have to pass the credential (key-pair and monotonic count) to any party who is delegated to make successive updates to the variable with the `EFI_VARIABLE_AUTHENTICATED_WRITE_ACCESS` set. This 3-tuple of {public key, private key, monotonic count} becomes part of the management metadata for these access-controlled items.

The responsibility of the caller that invokes the `SetVariable()` service with the `EFI_VARIABLE_AUTHENTICATED_WRITE_ACCESS` attribute will do the following prior to invoking the service:

- Update the Monotonic Count value.
- Hash the variable contents (Data, Size, Monotonic count) using the `HashType` in the `AuthInfo` structure.
- Sign the resultant hash of above step using a caller private key and create the digital signature `Signature`. Ensure that the public key associated with signing private key is in the `AuthInfo` structure.
- Invoke SetVariables with `EFI_VARIABLE_AUTHENTICATED_WRITE_ACCESS` attribute set.
The responsibility of the firmware that implements the SetVariable() service and supports the EFI_VARIABLE_AUTHENTICATED_WRITE_ACCESS attribute will do the following in response to being called:

- The first time it uses SetVariable with the EFI_VARIABLE_AUTHENTICATED_WRITE_ACCESS attribute set. Use the public key in the AuthInfo structure for subsequent verification.
- Hash the variable contents (Data, Size, Monotonic count) using the HashType in the AuthInfo structure.
- Compare the public key in the AuthInfo structure with the public key passed in on the first invocation.
- Verify the digital signature Signature of the signed hash using the stored public key associated with the variable.
- If comparison fails, return EFI_SECURITY_VIOLATION.
- Compare the new monotonic count and ensure that it is greater than the last SetVariable operation with the EFI_VARIABLE_AUTHENTICATED_WRITE_ACCESS attribute set.
- If new monotonic count is not strictly greater, then return EFI_SECURITY_VIOLATION.

NOTE: Special processing by SetVariable() for Secure Boot Mode variables and the Platform Key is defined in Section 32.3.

8.2.4 Hardware Error Record Persistence

This section defines how Hardware Error Record Persistence is to be implemented. By implementing support for Hardware Error Record Persistence, the platform enables the OS to utilize the EFI Variable Services to save hardware error records so they are persistent and remain available across OS sessions until they are explicitly cleared or overwritten by their creator.

8.2.4.1 Hardware Error Record Non-Volatile Store

A platform which implements support hardware error record persistence is required to guarantee some amount of NVR is available to the OS for saving hardware error records. The platform communicates the amount of space allocated for error records via the QueryVariableInfo routine as described in Appendix P.

8.2.4.2 Hardware Error Record Variables

This section defines a set of Hardware Error Record variables that have architecturally defined meanings. In addition to the defined data content, each such variable has an architecturally defined attribute that indicates when the data variable may be accessed. The variables with an attribute of HR are stored in the portion of NVR allocated for error records. NV, BS and RT have the meanings defined in section 3.2. All hardware error record variables use the EFI_HARDWARE_ERROR_VARIABLE VendorGuid:

```
#define EFI_HARDWARE_ERROR_VARIABLE
{0x414E6BDD,0xE47B,0x47cc,{0xB2,0x44,0xBB,0x61,0x02,0x0C,0xF5,0x16}}
```
Table 8-4 Hardware Error Record Persistence Variables

<table>
<thead>
<tr>
<th>Variable Name</th>
<th>Attribute</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>HwErrRec####</td>
<td>NV, BS, RT, HR</td>
<td>A hardware error record. #### is a printed hex value. No 0x or h is included in the hex value</td>
</tr>
</tbody>
</table>

The HwErrRec#### variable contains a hardware error record. Each HwErrRec#### variable is the name "HwErrRec" appended with a unique 4-digit hexadecimal number. For example, HwErrRec0001, HwErrRec0002, HwErrRecF31A, etc. The HR attribute indicates that this variable is to be stored in the portion of NVR allocated for error records.

8.2.4.3 Common Platform Error Record Format

Error record variables persisted using this interface are encoded in the Common Platform Error Record format, which is described in appendix N of the UEFI Specification. Because error records persisted using this interface conform to this standardized format, the error information may be used by entities other than the OS.

8.3 Time Services

This section contains function definitions for time-related functions that are typically needed by operating systems at runtime to access underlying hardware that manages time information and services. The purpose of these interfaces is to provide operating system writers with an abstraction for hardware time devices, thereby relieving the need to access legacy hardware devices directly. There is also a stalling function for use in the preboot environment. Table 8-5 lists the time services functions described in this section:

Table 8-5 Time Services Functions

<table>
<thead>
<tr>
<th>Name</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>GetTime</td>
<td>Runtime</td>
<td>Returns the current time and date, and the time-keeping capabilities of the platform.</td>
</tr>
<tr>
<td>SetTime</td>
<td>Runtime</td>
<td>Sets the current local time and date information.</td>
</tr>
<tr>
<td>GetWakeupTime</td>
<td>Runtime</td>
<td>Returns the current wakeup alarm clock setting.</td>
</tr>
<tr>
<td>SetWakeupTime</td>
<td>Runtime</td>
<td>Sets the system wakeup alarm clock time.</td>
</tr>
</tbody>
</table>

GetTime()

Summary

Returns the current time and date information, and the time-keeping capabilities of the hardware platform.
Prototype

```c
typedef EFI_STATUS
GetTime (  
    OUT EFI_TIME *Time,  
    OUT EFI_TIME_CAPABILITIES *Capabilities OPTIONAL  
);
```

Parameters

- **Time**: A pointer to storage to receive a snapshot of the current time. Type `EFI_TIME` is defined in “Related Definitions.”
- **Capabilities**: An optional pointer to a buffer to receive the real time clock device’s capabilities. Type `EFI_TIME_CAPABILITIES` is defined in “Related Definitions.”

Related Definitions

```
//**************************************************************
//EFI_TIME
//**************************************************************
// This represents the current time information
typedef struct {
    UINT16   Year;       // 1900 – 9999
    UINT8    Month;      // 1 – 12
    UINT8    Day;        // 1 – 31
    UINT8    Hour;       // 0 – 23
    UINT8    Minute;     // 0 – 59
    UINT8    Second;     // 0 – 59
    UINT8    Pad1;
    UINT32   Nanosecond; // 0 – 999,999,999
    INT16    TimeZone;   // -1440 to 1440 or 2047
    UINT8    Daylight;
    UINT8    Pad2;
} EFI_TIME;
```

```
//**************************************************************
// Bit Definitions for EFI_TIME.Daylight. See below.
//**************************************************************
#define EFI_TIME_ADJUST_DAYLIGHT  0x01
#define EFI_TIME_IN_DAYLIGHT     0x02
```

```
//**************************************************************
// Value Definition for EFI_TIME.TimeZone. See below.
//**************************************************************
#define EFI_UNSPECIFIED_TIMEZONE  0x07FF
```
Year, Month, Day  The current local date.

Hour, Minute, Second, Nanosecond

The current local time. Nanoseconds report the current fraction of a second in the device. The format of the time is `hh:mm:ss.nnnnnnnnn`. A battery backed real time clock device maintains the date and time.

TimeZone

The time's offset in minutes from UTC. If the value is `EFI_UNSPECIFIED_TIMEZONE`, then the time is interpreted as a local time. The `TimeZone` is the number of minutes that the local time is relative to UTC. To calculate the `TimeZone` value, follow this equation: `Localtime = UTC - TimeZone`.

To further illustrate this, an example is given below:

PST (Pacific Standard Time is 12PM) = UTC (8PM) - 8 hours (480 minutes)

In this case, the value for `TimeZone` would be 480 if referencing PST.

Daylight

A bitmask containing the daylight savings time information for the time.

The `EFI_TIME_ADJUST_DAYLIGHT` bit indicates if the time is affected by daylight savings time or not. This value does not indicate that the time has been adjusted for daylight savings time. It indicates only that it should be adjusted when the `EFI_TIME` enters daylight savings time.

If `EFI_TIME_IN_DAYLIGHT` is set, the time has been adjusted for daylight savings time.

All other bits must be zero.

When entering daylight saving time, if the time is affected, but hasn't been adjusted (DST = 1), use the new calculation: 1. The date/time should be increased by the appropriate amount. 2. The `TimeZone` should be decreased by the appropriate amount (EX: +480 changes to +420 when moving from PST to PDT). 3. The `Daylight` value changes to 3.

When exiting daylight saving time, if the time is affected and has been adjusted (DST = 3), use the new calculation: 1. The date/time should be decreased by the appropriate amount. 2. The `TimeZone` should be increased by the appropriate amount. 3. The `Daylight` value changes to 1.
// EFI_TIME_CAPABILITIES
/*------------------------------------------------------------------------------------------
// This provides the capabilities of the
// real time clock device as exposed through the EFI interfaces.
typedef struct {
    UINT32 Resolution;
    UINT32 Accuracy;
    BOOLEAN SetsToZero;
} EFI_TIME_CAPABILITIES;

Resolution
Provides the reporting resolution of the real-time clock device
in counts per second. For a normal PC-AT CMOS RTC device,
this value would be 1 Hz, or 1, to indicate that the device only
reports the time to the resolution of 1 second.

Accuracy
Provides the timekeeping accuracy of the real-time clock in an
error rate of 1E-6 parts per million. For a clock with an
accuracy of 50 parts per million, the value in this field would
be 50,000,000.

SetsToZero
A TRUE indicates that a time set operation clears the device’s
time below the Resolution reporting level. A FALSE indicates
that the state below the Resolution level of the device is not
cleared when the time is set. Normal PC-AT CMOS RTC
devices set this value to FALSE.

Description
The GetTime() function returns a time that was valid sometime during the call to the function. While
the returned EFI_TIME structure contains TimeZone and Daylight savings time information, the actual
clock does not maintain these values. The current time zone and daylight saving time information
returned by GetTime() are the values that were last set via SetTime().

The GetTime() function should take approximately the same amount of time to read the time each time
it is called. All reported device capabilities are to be rounded up.

During runtime, if a PC-AT CMOS device is present in the platform the caller must synchronize access to
the device before calling GetTime().
Status Codes Returned

<table>
<thead>
<tr>
<th>Code</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>EFI_SUCCESS</td>
<td>The operation completed successfully.</td>
</tr>
<tr>
<td>EFI_INVALID_PARAMETER</td>
<td><em>Time</em> is NULL.</td>
</tr>
<tr>
<td>EFI_DEVICE_ERROR</td>
<td>The time could not be retrieved due to a hardware error.</td>
</tr>
<tr>
<td>EFI_UNSUPPORTED</td>
<td>This call is not supported by this platform at the time the call is made.</td>
</tr>
<tr>
<td></td>
<td>The platform should describe this runtime service as unsupported at runtime</td>
</tr>
<tr>
<td></td>
<td>via an EFI_RT_PROPERTIES_TABLE configuration table.</td>
</tr>
</tbody>
</table>

SetTime()

Summary
Sets the current local time and date information.

Prototype

```c
typedef
EFI_STATUS
SetTime (  
    IN EFI_TIME *Time
);
```

Parameters

- **Time**: A pointer to the current time. Type EFI_TIME is defined in the GetTime() function description. Full error checking is performed on the different fields of the EFI_TIME structure (refer to the EFI_TIME definition in the GetTime() function description for full details), and EFI_INVALID_PARAMETER is returned if any field is out of range.

Description

The SetTime() function sets the real time clock device to the supplied time, and records the current time zone and daylight savings time information. The SetTime() function is not allowed to loop based on the current time. For example, if the device does not support a hardware reset for the sub-resolution time, the code is not to implement the feature by waiting for the time to wrap.

During runtime, if a PC-AT CMOS device is present in the platform the caller must synchronize access to the device before calling SetTime().
UEFI Specification, Version 2.9  Services — Runtime Services

Status Codes Returned

<table>
<thead>
<tr>
<th>Status Code</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>EFI_SUCCESS</td>
<td>The operation completed successfully.</td>
</tr>
<tr>
<td>EFI_INVALID_PARAMETER</td>
<td>A time field is out of range.</td>
</tr>
<tr>
<td>EFI_DEVICE_ERROR</td>
<td>The time could not be set due to a hardware error.</td>
</tr>
<tr>
<td>EFI_UNSUPPORTED</td>
<td>This call is not supported by this platform at the time the call is made.</td>
</tr>
<tr>
<td></td>
<td>The platform should describe this runtime service as unsupported at runtime</td>
</tr>
<tr>
<td></td>
<td>via an EFI_RT_PROPERTIES_TABLE configuration table.</td>
</tr>
</tbody>
</table>

GetWakeupTime()

Summary

Returns the current wakeup alarm clock setting.

Prototype

typedef
EFI_STATUS
GetWakeupTime (  
  OUT BOOLEAN *Enabled,  
  OUT BOOLEAN *Pending,  
  OUT EFI_TIME *Time  
);

Parameters

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Enabled</td>
<td>Indicates if the alarm is currently enabled or disabled.</td>
</tr>
<tr>
<td>Pending</td>
<td>Indicates if the alarm signal is pending and requires acknowledgement.</td>
</tr>
<tr>
<td>Time</td>
<td>The current alarm setting. Type EFI_TIME is defined in the GetTime() function description.</td>
</tr>
</tbody>
</table>

Description

The alarm clock time may be rounded from the set alarm clock time to be within the resolution of the alarm clock device. The resolution of the alarm clock device is defined to be one second.

During runtime, if a PC-AT CMOS device is present in the platform the caller must synchronize access to the device before calling GetWakeupTime().
Status Codes Returned

<table>
<thead>
<tr>
<th>Code</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>EFI_SUCCESS</td>
<td>The alarm settings were returned.</td>
</tr>
<tr>
<td>EFI_INVALID_PARAMETER</td>
<td>Enabled is NULL.</td>
</tr>
<tr>
<td>EFI_INVALID_PARAMETER</td>
<td>Pending is NULL.</td>
</tr>
<tr>
<td>EFI_INVALID_PARAMETER</td>
<td>Time is NULL.</td>
</tr>
<tr>
<td>EFI_DEVICE_ERROR</td>
<td>The wakeup time could not be retrieved due to a hardware error.</td>
</tr>
<tr>
<td>EFI_UNSUPPORTED</td>
<td>This call is not supported by this platform at the time the call is made. The platform should describe this runtime service as unsupported at runtime via an EFI_RT_PROPERTIES_TABLE configuration table.</td>
</tr>
</tbody>
</table>

SetWakeupTime()

Summary
Sets the system wakeup alarm clock time.

Prototype

```c
typedef
EFI_Status
SetWakeupTime (  
    IN BOOLEAN Enable,   
    IN EFI_TIME *Time OPTIONAL
);
```

Parameters

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Enable</td>
<td>Enable or disable the wakeup alarm. If Enable is TRUE, the time to set the wakeup alarm for. Type EFI_TIME is defined in the GetTime() function description. If Enable is FALSE, then this parameter is optional, and may be NULL.</td>
</tr>
<tr>
<td>Time</td>
<td>Time to set the wakeup alarm for.</td>
</tr>
</tbody>
</table>

Description

Setting a system wakeup alarm causes the system to wake up or power on at the set time. When the alarm fires, the alarm signal is latched until it is acknowledged by calling SetWakeupTime() to disable the alarm. If the alarm fires before the system is put into a sleeping or off state, since the alarm signal is latched the system will immediately wake up. If the alarm fires while the system is off and there is insufficient power to power on the system, the system is powered on when power is restored.

For an ACPI-aware operating system, this function only handles programming the wakeup alarm for the desired wakeup time. The operating system still controls the wakeup event as it normally would through the ACPI Power Management register set.

The resolution for the wakeup alarm is defined to be 1 second.
During runtime, if a PC-AT CMOS device is present in the platform the caller must synchronize access to the device before calling `SetWakeupTime()`.

**Status Codes Returned**

<table>
<thead>
<tr>
<th>Status Code</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>EFI_SUCCESS</td>
<td>If <code>Enable</code> is <code>TRUE</code>, then the wakeup alarm was enabled. If <code>Enable</code> is <code>FALSE</code>, then the wakeup alarm was disabled.</td>
</tr>
<tr>
<td>EFI_INVALID_PARAMETER</td>
<td>A time field is out of range.</td>
</tr>
<tr>
<td>EFI_DEVICE_ERROR</td>
<td>The wakeup time could not be set due to a hardware error.</td>
</tr>
<tr>
<td>EFI_UNSUPPORTED</td>
<td>This call is not supported by this platform at the time the call is made. The platform should describe this runtime service as unsupported at runtime via an <code>EFI_RT_PROPERTIES_TABLE</code> configuration table.</td>
</tr>
</tbody>
</table>

### 8.4 Virtual Memory Services

This section contains function definitions for the virtual memory support that may be optionally used by an operating system at runtime. If an operating system chooses to make EFI runtime service calls in a virtual addressing mode instead of the flat physical mode, then the operating system must use the services in this section to switch the EFI runtime services from flat physical addressing to virtual addressing. Table 8-6 lists the virtual memory service functions described in this section. The system firmware must follow the processor-specific rules outlined in Section 2.3.2 through Section 2.3.6 in the layout of the EFI memory map to enable the OS to make the required virtual mappings.

**Table 8-6 Virtual Memory Functions**

<table>
<thead>
<tr>
<th>Name</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>SetVirtualAddressMap</td>
<td>Runtime</td>
<td>Used by an OS loader to convert from physical addressing to virtual addressing.</td>
</tr>
<tr>
<td>ConvertPointer</td>
<td>Runtime</td>
<td>Used by EFI components to convert internal pointers when switching to virtual addressing.</td>
</tr>
</tbody>
</table>
**SetVirtualAddressMap()**

**Summary**
Changes the runtime addressing mode of EFI firmware from physical to virtual.

**Prototype**
```c
typedef EFI_STATUS
SetVirtualAddressMap (  
    IN UINTN MemoryMapSize,  
    IN UINTN DescriptorSize,  
    IN UINT32 DescriptorVersion,  
    IN EFI_MEMORY_DESCRIPTOR *VirtualMap
);
```

**Parameters**
- **MemoryMapSize** The size in bytes of `VirtualMap`.
- **DescriptorSize** The size in bytes of an entry in the `VirtualMap`.
- **DescriptorVersion** The version of the structure entries in `VirtualMap`.
- **VirtualMap** An array of memory descriptors which contain new virtual address mapping information for all runtime ranges. Type `EFI_MEMORY_DESCRIPTOR` is defined in the `EFI_BOOT_SERVICES.GetMemoryMap()` function description.

**Description**
The `SetVirtualAddressMap()` function is used by the OS loader. The function can only be called at runtime, and is called by the owner of the system’s memory map: i.e., the component which called `EFI_BOOT_SERVICES.ExitBootServices()`. All events of type `EVT_SIGNAL_VIRTUAL_ADDRESS_CHANGE` must be signaled before `SetVirtualAddressMap()` returns.

This call changes the addresses of the runtime components of the EFI firmware to the new virtual addresses supplied in the `VirtualMap`. The supplied `VirtualMap` must provide a new virtual address for every entry in the memory map at `ExitBootServices()` that is marked as being needed for runtime usage. All of the virtual address fields in the `VirtualMap` must be aligned on 4 KiB boundaries.

The call to `SetVirtualAddressMap()` must be done with the physical mappings. On successful return from this function, the system must then make any future calls with the newly assigned virtual mappings. All address space mappings must be done in accordance to the cacheability flags as specified in the original address map.

When this function is called, all events that were registered to be signaled on an address map change are notified. Each component that is notified must update any internal pointers for their new addresses. This can be done with the `ConvertPointer()` function. Once all events have been notified, the EFI firmware reapplies image “fix-up” information to virtually relocate all runtime images to their new addresses. In addition, all of the fields of the EFI Runtime Services Table except `SetVirtualAddressMap` and `ConvertPointer` must be converted from physical pointers to virtual pointers using the
ConvertPointer() service. The SetVirtualAddressMap() and ConvertPointer() services are only callable in physical mode, so they do not need to be converted from physical pointers to virtual pointers. Several fields of the EFI System Table must be converted from physical pointers to virtual pointers using the ConvertPointer() service. These fields include FirmwareVendor, RuntimeServices, and ConfigurationTable. Because contents of both the EFI Runtime Services Table and the EFI System Table are modified by this service, the 32-bit CRC for the EFI Runtime Services Table and the EFI System Table must be recomputed.

A virtual address map may only be applied one time. Once the runtime system is in virtual mode, calls to this function return EFI_UNSUPPORTED.

Status Codes Returned

<table>
<thead>
<tr>
<th>Status Code</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>EFI_SUCCESS</td>
<td>The virtual address map has been applied.</td>
</tr>
<tr>
<td>EFI_UNSUPPORTED</td>
<td>EFI firmware is not at runtime, or the EFI firmware is already in virtual</td>
</tr>
<tr>
<td></td>
<td>address mapped mode.</td>
</tr>
<tr>
<td>EFI_INVALID_PARAMETER</td>
<td>DescriptorSize or DescriptorVersion is invalid.</td>
</tr>
<tr>
<td>EFI_NO_MAPPING</td>
<td>A virtual address was not supplied for a range in the memory map that</td>
</tr>
<tr>
<td></td>
<td>requires a mapping.</td>
</tr>
<tr>
<td>EFI_NOT_FOUND</td>
<td>A virtual address was supplied for an address that is not found in the</td>
</tr>
<tr>
<td></td>
<td>memory map.</td>
</tr>
<tr>
<td>EFI_UNSUPPORTED</td>
<td>This call is not supported by this platform at the time the call is made.</td>
</tr>
<tr>
<td></td>
<td>The platform should describe this runtime service as unsupported at runtime</td>
</tr>
<tr>
<td></td>
<td>via an EFI_RT_PROPERTIES_TABLE configuration table.</td>
</tr>
</tbody>
</table>

ConvertPointer()

Summary
Determines the new virtual address that is to be used on subsequent memory accesses.

Prototype

```c
typedef
    EFI_STATUS
ConvertPointer (  
    IN UINTN    DebugDisposition,
    IN VOID    **Address
);
```

Parameters

- DebugDisposition Supplies type information for the pointer being converted. See “Related Definitions.”
- Address A pointer to a pointer that is to be fixed to be the value needed for the new virtual address mappings being applied.
Related Definitions

```c
//EFI_OPTIONAL_PTR
#define EFI_OPTIONAL_PTR 0x00000001
```

Description

The **ConvertPointer()** function is used by an EFI component during the SetVirtualAddressMap() operation. **ConvertPointer()** must be called using physical address pointers during the execution of SetVirtualAddressMap().

The **ConvertPointer()** function updates the current pointer pointed to by **Address** to be the proper value for the new address map. Only runtime components need to perform this operation. The **EFI_BOOT_SERVICES.CreateEvent()** function is used to create an event that is to be notified when the address map is changing. All pointers the component has allocated or assigned must be updated.

If the **EFI_OPTIONAL_PTR** flag is specified, the pointer being converted is allowed to be **NULL**.

Once all components have been notified of the address map change, firmware fixes any compiled in pointers that are embedded in any runtime image.

Status Codes Returned

<table>
<thead>
<tr>
<th>Code</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>EFI_SUCCESS</td>
<td>The pointer pointed to by <strong>Address</strong> was modified.</td>
</tr>
<tr>
<td>EFI_NOT_FOUND</td>
<td>The pointer pointed to by <strong>Address</strong> was not found to be part of the current memory map. This is normally fatal.</td>
</tr>
<tr>
<td>EFI_INVALID_PARAMETER</td>
<td><strong>Address</strong> is <strong>NULL</strong>.</td>
</tr>
<tr>
<td>EFI_INVALID_PARAMETER</td>
<td><em>Address</em> is <strong>NULL</strong> and <strong>DebugDisposition</strong> does not have the <strong>EFI_OPTIONAL_PTR</strong> bit set.</td>
</tr>
<tr>
<td>EFI_UNSUPPORTED</td>
<td>This call is not supported by this platform at the time the call is made. The platform should describe this runtime service as unsupported at runtime via an <strong>EFI_RT_PROPERTIES_TABLE</strong> configuration table.</td>
</tr>
</tbody>
</table>

8.5 Miscellaneous Runtime Services

This section contains the remaining function definitions for runtime services not defined elsewhere but which are required to complete the definition of the EFI environment. **Table 8-7** lists the Miscellaneous Runtime Services.
8.5.1 Reset System

This section describes the reset system runtime service and its associated data structures.

ResetSystem()

Summary

Resets the entire platform. If the platform supports `EFI_RESET_NOTIFICATION_PROTOCOL`, then prior to completing the reset of the platform, all of the pending notifications must be called.

Prototype

```c
typedef VOID (EFIAPI *EFI_RESET_SYSTEM) (IN EFI_RESET_TYPE ResetType, 
                                           IN EFI_STATUS ResetStatus, 
                                           IN UINTN DataSize, 
                                           IN VOID *ResetData OPTIONAL);
```

Parameters

- **ResetType** The type of reset to perform. Type `EFI_RESET_TYPE` is defined in “Related Definitions” below.
- **ResetStatus** The status code for the reset. If the system reset is part of a normal operation, the status code would be `EFI_SUCCESS`. If the system reset is due to some type of failure the most appropriate EFI Status code would be used.
- **DataSize** The size, in bytes, of `ResetData`.
- **ResetData** For a `ResetType` of `EfiResetCold`, `EfiResetWarm`, or `EfiResetShutdown` the data buffer starts with a Null-terminated string, optionally followed by additional binary data. The string is a description that the caller may use to further indicate the reason for the system reset. For a `ResetType` of `EfiResetPlatformSpecific` the data buffer also starts with a Null-terminated string that is followed by an EFI_GUID that describes the specific type of reset to perform.

Table 8-7 Miscellaneous Runtime Services

<table>
<thead>
<tr>
<th>Name</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>GetNextHighMonotonicCount</td>
<td>Runtime</td>
<td>Returns the next high 32 bits of the platform’s monotonic counter.</td>
</tr>
<tr>
<td>ResetSystem</td>
<td>Runtime</td>
<td>Resets the entire platform.</td>
</tr>
<tr>
<td>UpdateCapsule</td>
<td>Runtime</td>
<td>Pass capsules to the firmware. The firmware may process the capsules immediately or return a value to be passed into <code>ResetSystem()</code> that will cause the capsule to be processed by the firmware as part of the reset process.</td>
</tr>
<tr>
<td>QueryCapsuleCapabilities</td>
<td>Runtime</td>
<td>Returns if the capsule can be supported via <code>UpdateCapsule()</code></td>
</tr>
</tbody>
</table>
Related Definitions

```c
typedef enum {
    EfiResetCold,
    EfiResetWarm,
    EfiResetShutdown
} EFI_RESET_TYPE;
```

Description

The `ResetSystem()` function resets the entire platform, including all processors and devices, and reboots the system.

Calling this interface with `ResetType` of `EfiResetCold` causes a system-wide reset. This sets all circuitry within the system to its initial state. This type of reset is asynchronous to system operation and operates without regard to cycle boundaries. `EfiResetCold` is tantamount to a system power cycle.

Calling this interface with `ResetType` of `EfiResetWarm` causes a system-wide initialization. The processors are set to their initial state, and pending cycles are not corrupted. If the system does not support this reset type, then an `EfiResetCold` must be performed.

Calling this interface with `ResetType` of `EfiResetShutdown` causes the system to enter a power state equivalent to the ACPI G2/SS or G3 states. If the system does not support this reset type, then when the system is rebooted, it should exhibit the `EfiResetCold` attributes.

Calling this interface with `ResetType` of `EfiResetPlatformSpecific` causes a system-wide reset. The exact type of the reset is defined by the `EFI_GUID` that follows the Null-terminated Unicode string passed into `ResetData`. If the platform does not recognize the `EFI_GUID` in `ResetData` the platform must pick a supported reset type to perform. The platform may optionally log the parameters from any non-normal reset that occurs.

The `ResetSystem()` function does not return.

8.5.2 Get Next High Monotonic Count

This section describes the `GetNextHighMonotonicCount` runtime service and its associated data structures.

`GetNextHighMonotonicCount()`

Summary

Returns the next high 32 bits of the platform’s monotonic counter.
Prototype

```
typedef
EFI_STATUS
GetNextHighMonotonicCount (
   OUT UINT32   *HighCount
);
```

Parameters

- **HighCount** Pointer to returned value.

Description

The `GetNextHighMonotonicCount()` function returns the next high 32 bits of the platform’s monotonic counter.

The platform’s monotonic counter is comprised of two 32-bit quantities: the high 32 bits and the low 32 bits. During boot service time the low 32-bit value is volatile: it is reset to zero on every system reset and is increased by 1 on every call to `GetNextMonotonicCount()`. The high 32-bit value is nonvolatile and is increased by 1 whenever the system resets, whenever `GetNextHighMonotonicCount()` is called, or whenever the low 32-bit count (returned by `GetNextMonotonicCount()`) overflows.

The `EFI_BOOT_SERVICES.GetNextMonotonicCount()` function is only available at boot services time. If the operating system wishes to extend the platform monotonic counter to runtime, it may do so by utilizing `GetNextHighMonotonicCount()`. To do this, before calling `EFI_BOOT_SERVICES.ExitBootServices()` the operating system would call `GetNextMonotonicCount()` to obtain the current platform monotonic count. The operating system would then provide an interface that returns the next count by:

- Adding 1 to the last count.
- Before the lower 32 bits of the count overflows, call `GetNextHighMonotonicCount()`. This will increase the high 32 bits of the platform’s nonvolatile portion of the monotonic count by 1.

This function may only be called at Runtime.

Status Codes Returned

<table>
<thead>
<tr>
<th>Status Code</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>EFI_SUCCESS</td>
<td>The next high monotonic count was returned.</td>
</tr>
<tr>
<td>EFI_DEVICE_ERROR</td>
<td>The device is not functioning properly.</td>
</tr>
<tr>
<td>EFI_INVALID_PARAMETER</td>
<td><code>HighCount</code> is <strong>NULL</strong>.</td>
</tr>
<tr>
<td>EFI_UNSUPPORTED</td>
<td>This call is not supported by this platform at the time the call is made. The platform should describe this runtime service as unsupported at runtime via an <strong>EFI_RT_PROPERTIES_TABLE</strong> configuration table.</td>
</tr>
</tbody>
</table>
8.5.3 Update Capsule

This runtime function allows a caller to pass information to the firmware. Update Capsule is commonly used to update the firmware FLASH or for an operating system to have information persist across a system reset.

**UpdateCapsule()**

**Summary**

Passes capsules to the firmware with both virtual and physical mapping. Depending on the intended consumption, the firmware may process the capsule immediately. If the payload should persist across a system reset, the reset value returned from `EFI_QueryCapsuleCapabilities` must be passed into `ResetSystem()` and will cause the capsule to be processed by the firmware as part of the reset process.

**Prototype**

```c
typedef
  EFI_STATUS
UpdateCapsule (  
  IN EFI_CAPSULE_HEADER **CapsuleHeaderArray,  
  IN UINTN CapsuleCount,  
  IN EFI_PHYSICAL_ADDRESS ScatterGatherList OPTIONAL  
);
```

**Parameters**

- **CapsuleHeaderArray** Virtual pointer to an array of virtual pointers to the capsules being passed into update capsule. Each capsules is assumed to stored in contiguous virtual memory. The capsules in the `CapsuleHeaderArray` must be the same capsules as the `ScatterGatherList`. The `CapsuleHeaderArray` must have the capsules in the same order as the `ScatterGatherList`.

- **CapsuleCount** Number of pointers to `EFI_CAPSULE_HEADER` in `CapsuleHeaderArray`.

- **ScatterGatherList** Physical pointer to a set of `EFI_CAPSULE_BLOCK_DESCRIPTOR` that describes the location in physical memory of a set of capsules. See "Related Definitions" for an explanation of how more than one capsule is passed via this interface. The capsules in the `ScatterGatherList` must be in the same order as the `CapsuleHeaderArray`. This parameter is only referenced if the capsules are defined to persist across system reset.
Related Definitions

```c
typedef struct {
    UINT64 Length;
    union {
        EFI_PHYSICAL_ADDRESS DataBlock;
        EFI_PHYSICAL_ADDRESS ContinuationPointer;
    } Union;
} EFI_CAPSULE_BLOCK_DESCRIPTOR;
```

- **Length**: Length in bytes of the data pointed to by DataBlock/ContinuationPointer.
- **DataBlock**: Physical address of the data block. This member of the union is used if Length is not equal to zero.
- **ContinuationPointer**: Physical address of another block of EFI_CAPSULE_BLOCK_DESCRIPTOR structures. This member of the union is used if Length is equal to zero. If ContinuationPointer is zero this entry represents the end of the list.

This data structure defines the ScatterGatherList list the OS passes to the firmware. ScatterGatherList represents an array of structures and is terminated with a structure member whose Length is 0 and DataBlock physical address is 0. If Length is 0 and DataBlock physical address is not 0, the specified physical address is known as a “continuation pointer” and it points to a further list of EFI_CAPSULE_BLOCK_DESCRIPTOR structures. A continuation pointer is used to allow the scatter gather list to be contained in physical memory that is not contiguous. It also is used to allow more than a single capsule to be passed at one time.

```c
typedef struct {
    EFI_GUID CapsuleGuid;
    UINT32 HeaderSize;
    UINT32 Flags;
    UINT32 CapsuleImageSize;
} EFI_CAPSULE_HEADER;
```

- **CapsuleGuid**: A GUID that defines the contents of a capsule.
- **HeaderSize**: The size of the capsule header. This may be larger than the size of the EFI_CAPSULE_HEADER since CapsuleGuid may imply extended header entries.
- **Flags**: The Flags[15:0] bits are defined by CapsuleGuid. Flags[31:16] are defined by this specification.
- **CapsuleImageSize**: Size in bytes of the capsule (including capsule header).
#define  CAPSULE_FLAGS_PERSIST_ACROSS_RESET    0x00010000
#define  CAPSULE_FLAGS_POPULATE_SYSTEM_TABLE   0x00020000
#define  CAPSULE_FLAGS_INITIATE_RESET          0x00040000

Note:  A capsule which has the CAPSULE_FLAGS_INITIATE_RESET Flag must have
        CAPSULE_FLAGS_PERSIST_ACROSS_RESET set in its header as well. Firmware that encounters
        a capsule which has the CAPSULE_FLAGS_INITIATE_RESET Flag set in its header will initiate a
        reset of the platform which is compatible with the passed-in capsule request and will not return
        back to the caller.

typedef struct {
    UINT32    CapsuleArrayNumber;
    VOID*     CapsulePtr[1];
}  EFI_CAPSULE_TABLE;

CapsuleArrayNumber    The number of entries in the array of capsules.
CapsulePtr            A pointer to an array of capsules that contain the same
                       CapsuleGuid value. Each CapsulePtr points to an instance
                       of an EFI_CAPSULE_HEADER, with the capsule data
                       concatenated on its end.

Description

The UpdateCapsule() function allows the operating system to pass information to firmware. The
UpdateCapsule() function supports passing capsules in operating system virtual memory back to
firmware. Each capsule is contained in a contiguous virtual memory range in the operating system, but
both a virtual and physical mapping for the capsules are passed to the firmware.

If a capsule has the CAPSULE_FLAGS_PERSIST_ACROSS_RESET Flag set in its header, the firmware will
process the capsules after system reset. The caller must ensure to reset the system using the required
reset value obtained from QueryCapsuleCapabilities. If this flag is not set, the firmware will process the
capsules immediately.

A capsule which has the CAPSULE_FLAGS_POPULATE_SYSTEM_TABLE Flag must have
CAPSULE_FLAGS_PERSIST_ACROSS_RESET set in its header as well. Firmware that processes a
capsule that has the CAPSULE_FLAGS_POPULATE_SYSTEM_TABLE Flag set in its header will coalesce
the contents of the capsule from the ScatterGatherList into a contiguous buffer and must then place a
pointer to this coalesced capsule in the EFI System Table after the system has been reset. Agents
searching for this capsule will look in the EFI_CONFIGURATION_TABLE and search for the capsule’s
GUID and associated pointer to retrieve the data after the reset.
Table 8-8 Flag Firmware Behavior

<table>
<thead>
<tr>
<th>Flags</th>
<th>Firmware Behavior</th>
</tr>
</thead>
<tbody>
<tr>
<td>No Specification defined flags</td>
<td>Firmware attempts to immediately processes or launch the capsule. If capsule is not recognized, can expect an error.</td>
</tr>
<tr>
<td>CAPSULE_FLAGS_PERSIST ACROSS_RESET</td>
<td>Firmware will attempt to process or launch the capsule across a reset. If capsule is not recognized, can expect an error.</td>
</tr>
<tr>
<td>CAPSULE_FLAGS_PERSIST ACROSS_RESET +</td>
<td>Firmware will coalesce the capsule from the ScatterGatherList into a contiguous buffer and place a pointer to the coalesced capsule in the EFI System Table. Platform recognition of the capsule type is not required. If the action requires a reset which is unsupported by the platform, expect an error.</td>
</tr>
<tr>
<td>CAPSULE_FLAGS_PERSIST ACROSS_RESET +</td>
<td>Firmware will attempt to process or launch the capsule across a reset. The firmware will initiate a reset which is compatible with the passed-in capsule request and will not return back to the caller. If the capsule is not recognized, can expect an error. If the processing requires a reset which is unsupported by the platform, expect an error.</td>
</tr>
<tr>
<td>CAPSULE_FLAGS_PERSIST ACROSS_RESET +</td>
<td>The firmware will initiate a reset which is compatible with the passed-in capsule request and not return back to the caller. Upon resetting, the firmware will coalesce the capsule from the ScatterGatherList into a contiguous buffer and place a pointer to the coalesced capsule in the EFI System Table. Platform recognition of the capsule type is not required. If the action requires a reset which is unsupported by the platform, expect an error.</td>
</tr>
</tbody>
</table>

The EFI System Table entry must use the GUID from the CapsuleGuid field of the EFI_CAPSULE_HEADER. The EFI System Table entry must point to an array of capsules that contain the same CapsuleGuid value. The array must be prefixed by a UINT32 that represents the size of the array of capsules.

The set of capsules is pointed to by ScatterGatherList and CapsuleHeaderArray so the firmware will know both the physical and virtual addresses of the operating system allocated buffers. The scatter-gather list supports the situation where the virtual address range of a capsule is contiguous, but the physical addresses are not.

On architectures where the processor’s view of main memory is incoherent with the caches when the memory management unit is disabled, callers to UpdateCapsule() must perform cache maintenance to main memory on each ScatterGatherList element before calling UpdateCapsule(). This requirement only applies after the OS has called ExitBootServices().

If any of the capsules that are passed into this function encounter an error, the entire set of capsules will not be processed and the error encountered will be returned to the caller.
Status Codes Returned

<table>
<thead>
<tr>
<th>Status Code</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>EFI_SUCCESS</td>
<td>Valid capsule was passed. If CAPSULE_FLAGS_PERSIST_ACROSS_RESET is not set, the capsule has been successfully processed by the firmware.</td>
</tr>
<tr>
<td>EFI_INVALID_PARAMETER</td>
<td>CapsuleSize, or an incompatible set of flags were set in the capsule header.</td>
</tr>
<tr>
<td>EFI_INVALID_PARAMETER</td>
<td>CapsuleCount is 0</td>
</tr>
<tr>
<td>EFI_DEVICE_ERROR</td>
<td>The capsule update was started, but failed due to a device error.</td>
</tr>
<tr>
<td>EFI_UNSUPPORTED</td>
<td>The capsule type is not supported on this platform.</td>
</tr>
<tr>
<td>EFI_OUT_OF_RESOURCES</td>
<td>When ExitBootServices() has been previously called this error indicates the capsule is compatible with this platform but is not capable of being submitted or processed in runtime. The caller may resubmit the capsule prior to ExitBootServices().</td>
</tr>
<tr>
<td>EFI_OUT_OF_RESOURCES</td>
<td>When ExitBootServices() has not been previously called then this error indicates the capsule is compatible with this platform but there are insufficient resources to process.</td>
</tr>
<tr>
<td>EFI_UNSUPPORTED</td>
<td>This call is not supported by this platform at the time the call is made. The platform should describe this runtime service as unsupported at runtime via an EFI_RT_PROPERTIES_TABLE configuration table.</td>
</tr>
</tbody>
</table>

8.5.3.1 Capsule Definition

A capsule is simply a contiguous set of data that starts with an EFI_CAPSULE_HEADER. The CapsuleGuid field in the header defines the format of the capsule.

The capsule contents are designed to be communicated from an OS-present environment to the system firmware. To allow capsules to persist across system reset, a level of indirection is required for the description of a capsule, since the OS primarily uses virtual memory and the firmware at boot time uses physical memory. This level of abstraction is accomplished via the EFI_CAPSULE_BLOCK_DESCRIPTOR. The EFI_CAPSULE_BLOCK_DESCRIPTOR allows the OS to allocate contiguous virtual address space and describe this address space to the firmware as a discontinuous set of physical address ranges. The firmware is passed both physical and virtual addresses and pointers to describe the capsule so the firmware can process the capsule immediately or defer processing of the capsule until after a system reset.

In most instruction sets and OS architecture, allocation of physical memory is possible only on a “page” granularity (which can range for 4 KiB to at least 1 MiB). The EFI_CAPSULE_BLOCK_DESCRIPTOR must have the following properties to ensure the safe and well defined transition of the data:

- Each new capsule must start on a new page of memory.
- All pages except for the last must be completely filled by the capsule.
  - It is legal to pad the header to make it consume an entire page of data to enable the passing of page aligned data structures via a capsule. The last page must have at least one byte of capsule in it.
- Pages must be naturally aligned
- Pages may not overlap on another
• Firmware may never make an assumption about the page sizes the operating system is using. Multiple capsules can be concatenated together and passed via a single call to \texttt{UpdateCapsule()}. The physical address description of capsules are concatenated by converting the terminating \texttt{EFI\_CAPSULE\_BLOCK\_DESCRIPTOR} entry of the 1\textsuperscript{st} capsule into a continuation pointer by making it point to the \texttt{EFI\_CAPSULE\_BLOCK\_DESCRIPTOR} that represents the start of the 2\textsuperscript{nd} capsule. There is only a single terminating \texttt{EFI\_CAPSULE\_BLOCK\_DESCRIPTOR} entry and it is at the end of the last capsule in the chain.

The following algorithm must be used to find multiple capsules in a single scatter gather list:

• Look at the capsule header to determine the size of the capsule
  — The first Capsule header is always pointed to by the first \texttt{EFI\_CAPSULE\_BLOCK\_DESCRIPTOR} entry

• Walk the \texttt{EFI\_CAPSULE\_BLOCK\_DESCRIPTOR} list keeping a running count of the size each entry represents.

• If the \texttt{EFI\_CAPSULE\_BLOCK\_DESCRIPTOR} entry is a continuation pointer and the running current capsule size count is greater than or equal to the size of the current capsule this is the start of the next capsule.

• Make the new capsules the current capsule and repeat the algorithm.
Figure 8-1 shows a Scatter-Gather list of EFI_CAPSULE_BLOCK_DESCRIPTOR structures that describes two capsules. The left side of the figure shows OS view of the capsules as two separate contiguous virtual memory buffers. The center of the figure shows the layout of the data in system memory. The right hand side of the figure shows the ScatterGatherList list passed into the firmware. Since there are two capsules two independent EFI_CAPSULE_BLOCK_DESCRIPTOR lists exist that were joined together via a continuation pointer in the first list.
EFI_MEMORY_RANGE_CAPSULE_GUID

This capsule structure definition provides a means by which a third-party component (e.g. OS) can describe to firmware what regions in memory should be left untouched across the next reset.

Support for this capsule is optional. For platforms that support this capsule, they must advertise EFI_MEMORY_RANGE_CAPSULE in the EFI Configuration table using the EFI_MEMORY_RANGE_CAPSULE_GUID as the GUID in the GUID-pointer pair.

```
#define EFI_MEMORY_RANGE_CAPSULE_GUID

// {0DE9F0EC-88B6-428F-977A-258F1D0E5E72}
#endif
```

A memory range descriptor.

```c
typedef struct
    EFI_PHYSICAL_ADDRESS Address;
    UINT64 Length;
} EFI_MEMORY_RANGE;
```

**Address**

Physical address of memory location being described.

**Length**

Length in bytes.

The capsule descriptor that describes the memory ranges a platform firmware should leave untouched.

```c
typedef struct {
    EFI_CAPSULE_HEADER Header;
    UINT32 OsRequestedMemoryType;
    UINT64 NumberofMemoryRanges;
    EFI_MEMORY_RANGE MemoryRanges[];
} EFI_MEMORY_RANGE_CAPSULE;
```

**Header**

Header.CapsuleGuid = EFI_MEMORY_RANGE_CAPSULE_GUID
Header.Flags = CAPSULE_FLAGS_PERSIST_ACROSS_RESET

**OsRequestedMemoryType**

Must be in the 0x80000000-0xFFFFFFFF range
When UEFI Firmware processes the capsule, contents described in MemoryRanges[] will show up as OsRequestedMemoryType values in the EFI Memory Map.

**NumberofMemoryRanges**

Number of MemoryRanges[] entries. Must be a value of 1 or greater.

**MemoryRanges[]**

An array of memory ranges. Equivalent to MemoryRanges[NumberofMemoryRanges].
For a platform that intends to support the `EFI_MEMORY_RANGE_CAPSULE`, it must advertise `EFI_MEMORY_RANGE_CAPSULE_RESULT` in the EFI Configuration table using the `EFI_MEMORY_RANGE_CAPSULE_GUID` as the GUID in the GUID/pointer pair.

```c
typedef struct {
    UINT64 FirmwareMemoryRequirement;
    UINT64 NumberOfMemoryRanges;
} EFI_MEMORY_RANGE_CAPSULE_RESULT
```

- **FirmwareMemoryRequirement**
  - The maximum amount of memory in bytes that the UEFI firmware requires to initialize.

- **NumberOfMemoryRanges**
  - Will be 0 if no `EFI_MEMORY_RANGE_CAPSULE` has been processed.
  - If a `EFI_MEMORY_RANGE_CAPSULE` was processed, this number will be identical to the `EFI_MEMORY_RANGE_CAPSULE`.NumberOfMemoryRanges value.

### QueryCapsuleCapabilities()

**Summary**

Returns if the capsule can be supported via `UpdateCapsule()`.

**Prototype**

```c
typedef EFI_STATUS
QueryCapsuleCapabilities (  
    IN EFI_CAPSULE_HEADER **CapsuleHeaderArray,
    IN UINTN CapsuleCount,
    OUT UINT64 *MaximumCapsuleSize,
    OUT EFI_RESET_TYPE *ResetType
);  
```

- **CapsuleHeaderArray**
  - Virtual pointer to an array of virtual pointers to the capsules being passed into update capsule. The capsules are assumed to stored in contiguous virtual memory.

- **CapsuleCount**
  - Number of pointers to `EFI_CAPSULE_HEADER` in `CapsuleHeaderArray`.

- **MaximumCapsuleSize**
  - On output the maximum size in bytes that `UpdateCapsule()` can support as an argument to `UpdateCapsule()` via `CapsuleHeaderArray` and `ScatterGatherList`. Undefined on input.

- **ResetType**
  - Returns the type of reset required for the capsule update. Undefined on input.
Description

The `QueryCapsuleCapabilities()` function allows a caller to test to see if a capsule or capsules can be updated via `UpdateCapsule()`. The Flags values in the capsule header and size of the entire capsule is checked.

If the caller needs to query for generic capsule capability a fake `EFI_CAPSULE_HEADER` can be constructed where `CapsuleImageSize` is equal to `HeaderSize` that is equal to `sizeof(EFI_CAPSULE_HEADER)`. To determine reset requirements, `CAPSULE_FLAGS_PERSIST_ACROSS_RESET` should be set in the `Flags` field of the `EFI_CAPSULE_HEADER`.

Status Codes Returned

<table>
<thead>
<tr>
<th>Status Code</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>EFI_SUCCESS</td>
<td>Valid answer returned.</td>
</tr>
<tr>
<td>EFI_INVALID_PARAMETER</td>
<td><code>MaximumCapsuleSize</code> is <code>NULL</code>.</td>
</tr>
<tr>
<td>EFI_UNSUPPORTED</td>
<td>The capsule type is not supported on this platform, and <code>MaximumCapsuleSize</code> and <code>ResetType</code> are undefined.</td>
</tr>
<tr>
<td>EFI_OUT_OF_RESOURCES</td>
<td>When <code>ExitBootServices()</code> has been previously called this error indicates the capsule is compatible with this platform but is not capable of being submitted or processed in runtime. The caller may resubmit the capsule prior to <code>ExitBootServices()</code>.</td>
</tr>
<tr>
<td>EFI_OUT_OF_RESOURCES</td>
<td>When <code>ExitBootServices()</code> has not been previously called then this error indicates the capsule is compatible with this platform but there are insufficient resources to process.</td>
</tr>
<tr>
<td>EFI_UNSUPPORTED</td>
<td>This call is not supported by this platform at the time the call is made. The platform should describe this runtime service as unsupported at runtime via an <code>EFI_RT_PROPERTIES_TABLE</code> configuration table.</td>
</tr>
</tbody>
</table>

8.5.4 Exchanging information between the OS and Firmware

The firmware and an Operating System may exchange information through the `OsIndicationsSupported` and the `OSIndications` variables as follows:

- The `OsIndications` variable returns a UINT64 bitmask owned by the OS and is used to indicate which features the OS wants firmware to enable or which actions the OS wants the firmware to take. The OS will supply this data with a `SetVariable()` call.

- The `OsIndicationsSupported` variable returns a UINT64 bitmask owned by the firmware and indicates which of the OS indication features and actions that the firmware supports. This variable is recreated by firmware every boot, and cannot be modified by the OS.

The `EFI_OS_INDICATIONS_BOOT_TO_FW_UI` bit can be set in the `OsIndicationsSupported` variable by the firmware, if the firmware supports OS requests to stop at a firmware user interface. The `EFI_OS_INDICATIONS_BOOT_TO_FW_UI` bit can be set by the OS in the `OsIndications` variable, if the OS desires for the firmware to stop at a firmware user interface on the next boot. Once the firmware consumes this bit in the `OsIndications` variable and stops at the firmware user interface, the firmware should clear the bit from the `OsIndications` variable in order to acknowledge to the OS that the information was consumed and, more importantly, to prevent the firmware user interface from showing again on subsequent boots.
The **EFI_OS_INDICATIONS_TIMESTAMP_REVOCATION** bit can be set in the OsIndicationsSupported variable by the firmware, if the firmware supports timestamp based revocation and the "dbt" authorized timestamp database variable.

The **EFI_OS_INDICATIONS_FMP_CAPSULE_SUPPORTED** bit is set in OsIndicationsSupported variable if platform supports processing of Firmware Management Protocol update capsule as defined in Section 23.2. If set in OsIndications variable, the **EFI_OS_INDICATIONS_FMP_CAPSULE_SUPPORTED** bit has no function and is cleared on the next reboot.

The **EFI_OS_INDICATIONS_FILE_CAPSULE_DELIVERY_SUPPORTED** bit in OsIndicationsSupported variable is set if platform supports processing of file capsules per Section 8.5.5.

When submitting capsule via the Mass Storage Device method of Section 8.5.5, the bit **EFI_OS_INDICATIONS_FILE_CAPSULE_DELIVERY_SUPPORTED** in OsIndications variable must be set by submitter to trigger processing of submitted capsule on next reboot. This bit will be cleared from OsIndications by system firmware in all cases during processing following reboot.

The **EFI_OS_INDICATIONS_CAPSULE_RESULT_VAR_SUPPORTED** bit is set in OsIndicationsSupported variable if platform supports reporting of deferred capsule processing by creation of result variable as defined in Section 8.5.6. This bit has no function if set in OsIndications.

The **EFI_OS_INDICATIONS_START_OS_RECOVERY** bit is set in the OsIndicationsSupported variable if the platform supports both the ability for an OS to indicate that OS-defined recovery should commence upon reboot, as well as support for the short-form File Path Media Device Path (see Section 3.1.2). If this bit is set in OsIndications, the platform firmware must bypass processing of the BootOrder variable during boot, and skip directly to OS-defined recovery (see Section 3.4.1) followed by Platform-defined recovery (see Section 3.4.2). System firmware must clear this bit in OsIndications when it starts OS-defined recovery.

The **EFI_OS_INDICATIONS_START_PLATFORM_RECOVERY** bit is set in the OsIndicationsSupported variable if the platform supports both the ability for an OS to indicate that Platform-defined recovery should commence upon reboot, as well as support for the short-form File Path Media Device Path (see Section 3.1.2). If this bit is set in OsIndications, the platform firmware must bypass processing of the BootOrder variable during boot, and skip directly to platform-defined recovery (see Section 3.4.2). System firmware must clear this bit in OsIndications when it starts Platform-defined recovery.

In all cases, if either of **EFI_OS_INDICATIONS_START_OS_RECOVERY** or **EFI_OS_INDICATIONS_START_PLATFORM_RECOVERY** is set in OsIndicationsSupported, both must be set and supported.

The **EFI_OS_INDICATIONS_JSON_CONFIG_DATA_REFRESH** bit is set in the OsIndications variable by submitter to trigger collecting current configuration and reporting the refreshed data to EFI System Configuration Table on next boot. If not set, platform will not collect current configuration but report the cached configuration data to EFI System Configuration Table. The configuration data shall be installed to EFI System Configuration Table using the format of **EFI_JSON_CAPSULE_CONFIG_DATA** defined in Section 23.5.2. This bit will be cleared from OsIndications by system firmware once the refreshed data is reported.
If set in the `OsIndicationsSupported` variable, the `EFI_OS_INDICATIONS_JSON_CONFIG_DATA_REFRESH` bit has no function and is cleared on the next reboot.

### Related Definitions

```c
#define EFI_OS_INDICATIONS_BOOT_TO_FW_UI                0x0000000000000001
#define EFI_OS_INDICATIONS_TIMESTAMP_REVOCATION        0x0000000000000002
#define EFI_OS_INDICATIONS_FILE_CAPSULE_DELIVERY_SUPPORTED 0x0000000000000004
#define EFI_OS_INDICATIONS_FMP_CAPSULE_SUPPORTED       0x0000000000000008
#define EFI_OS_INDICATIONS_CAPSULE_RESULT_VAR_SUPPORTED 0x0000000000000010
#define EFI_OS_INDICATIONS_START_OS_RECOVERY           0x0000000000000020
#define EFI_OS_INDICATIONS_START_PLATFORM_RECOVERY     0x0000000000000040
#define EFI_OS_INDICATIONS_JSON_CONFIG_DATA_REFRESH    0x0000000000000080
```

### 8.5.5 Delivery of Capsules via file on Mass Storage device

As an alternative to the `UpdateCapsule()` runtime API, capsules of any type supported by platform may also be delivered to firmware via a file within the EFI system partition on the mass storage device targeted for boot. Capsules staged using this method are processed on the next system restart. This method is only available when booting from mass storage devices which are formatted with GPT (Section 5.3) and contain an EFI System Partition in the device image. System firmware will search for capsule when `EFI_OS_INDICATIONS_FILE_CAPSULE_DELIVERY_SUPPORTED` bit in `OsIndications` is set as described in Section 8.5.4.

The directory `\EFI\UpdateCapsule` (letter case ignored) within the active EFI System Partition is defined for delivery of capsule to firmware. The binary structure of a capsule file on mass storage device is identical to the contents of capsule delivered via the EFI RunTime API except that fragmentation using `EFI_CAPSULE_BLOCK_DESCRIPTOR` is not supported and the single capsule must be stored in contiguous bytes within the file starting with `EFI_CAPSULE_HEADER`. The size of the file must equal `CapsuleImageSize` or error will be generated and the capsule ignored. Only a single capsule with a single `EFI_CAPSULE_HEADER` may be submitted within a file but more than one file each containing a capsule may be submitted during a single restart.

The file name of the capsule shall be chosen by submitter using 8-bit ASCII characters appropriate to the file system of the EFI system partition (Section 13.3.1). After examination and processing of a file placed in this directory the file will (if possible) be deleted by firmware. The deletion is performed in case of successful processing and also in the case of error but failure to successfully delete is not itself a reportable error.

More than one capsule file each containing a single capsule image may be stored in the specified directory. In case of multiple files, the system firmware shall process files in alphabetical order using sort based on CHAR16 numerical value of file name characters, compared left to right. Lower case letter
characters will be converted to upper case before compare. When comparing file names of unequal length, the space character shall be used to pad shorter file names. In case of file name containing one or more period characters (.), the right-most period, and the text to the right of the right-most period in the file name, will be removed before compare. In case of any file names with identical text after excluding any text after the right-most period, the order of processing shall be determined by sorting of any text found to right of the right-most period in file name string.

If a capsule processing is terminated by error any remaining additional capsule files will be processed normally.

The directory \EFI\UpdateCapsule is checked for capsules only within the EFI system partition on the device specified in the active boot option determine by reference to BootNext variable or BootOrder variable processing. The active Boot Variable is the variable with highest priority BootNext or within BootOrder that refers to a device found to be present. Boot variables in BootOrder but referring to devices not present are ignored when determining active boot variable.

The device to be checked for \EFI\UpdateCapsule is identified by reference to FilePathList field within the selected active Boot#### variable. The system firmware is not required to check mass storage devices that do not contain boot target that is highest priority for boot nor to check a second EFI system partition not the target of the active boot variable.

In all cases that a capsule is identified for processing the system is restarted after capsule processing is completed. In case where BootNext variable was set, this variable is cleared when capsule processing is performed without actual boot of the variable indicated.

8.5.6 UEFI variable reporting on the Success or any Errors encountered in processing of capsules after restart

In cases where the processing of capsules is (1) delivered by call to UpdateCapsule() API but deferred to next restart, or (2) when capsules are delivered via mass storage device, a UEFI variable is created by firmware to indicate to capsule provider the status of the capsule processing. In the case were multiple capsules are delivered in calls to UpdateCapsule(), or multiple files on disk as described in Section 8.5.5, or when a capsule contains multiple payloads as described in Section 23.2, a separate result variable will be created for each capsule payload processed. The firmware will over-write result variables when calculated variable name already exists. However, to avoid unnecessarily consuming system variable store the result variable should be deleted by capsule provider after result status is examined.

UEFI variable reports will not be used when the entirety of capsule processing occurs within the call to UpdateCapsule() function.

The reporting variable attributes will be EFI_VARIABLE_NON_VOLATILE + EFI_VARIABLE_BOOTSERVICE_ACCESS + EFI_VARIABLE_RUNTIME_ACCESS.

The Vendor GUID of the reporting variable will be EFI_CAPSULE_REPORT_GUID. The name of the reporting variable will be CapsuleNNNN where NNNN is 4-digit hex number chosen by the firmware. The values of NNNN will be incremented by firmware starting at Capsule0000 and continuing up to the platform-defined maximum.

The platform will publish the platform maximum in a read-only variable named EFI_CAPSULE_REPORT_GUID:CapsuleMax. The contents of CapsuleMax will be the string "CapsuleNNNN" where NNNN is the highest value used by platform before rolling over to
Capsule0000. The platform will also publish the name of the last variable created in EFI_CAPSULE_REPORT_GUID: CapsuleLast.

When creating a new result variable, any previous variable with the same name will be overwritten. In case where variable storage is limited system firmware may optionally delete oldest report variables to create free space. If sufficient variable space cannot be freed the variable is not created.

<table>
<thead>
<tr>
<th>Variable Name</th>
<th>Attributes</th>
<th>Internal Format</th>
</tr>
</thead>
<tbody>
<tr>
<td>Capsule0000, Capsule0001, ... up to max</td>
<td>NV, BS, RT</td>
<td>EFI_CAPSULE_RESULT_VARIABLE</td>
</tr>
<tr>
<td>CapsuleMax</td>
<td>BS, RT, Read-Only</td>
<td>CHAR16[11] (no zero terminator)</td>
</tr>
<tr>
<td>CapsuleLast</td>
<td>NV, BS, RT, Read-Only</td>
<td>CHAR16[11] (no zero terminator)</td>
</tr>
</tbody>
</table>

**EFI_CAPSULE_REPORT_GUID**

```c
// {39B68C46-F7FB-441B-B6EC-16B0F69821F3}
#define EFI_CAPSULE_REPORT_GUID
    { 0x39b68c46, 0xf7fb, 0x441b, 
      {0xb6, 0xec, 0x16, 0xb0, 0xf6, 0x98, 0x21, 0xf3 }};
```

**Structure of the Capsule Processing Result Variable**

The Capsule Processing Result Variable contents always begin with the EFI_CAPSULE_RESULT_VARIABLE_HEADER structure. The value of CapsuleGuid determines any additional data that may follow within the instance of the Result Variable contents. For some values of CapsuleGuid no additional data may be defined.

As noted below, VariableTotalSize is the size of complete result variable including the entire header and any additional data required for particular CapsuleGuid types.

```c
typedef struct {
    UINT32 VariableTotalSize;
    UINT32 Reserved; //for alignment
    EFI_GUID CapsuleGuid;
    EFI_TIME CapsuleProcessed;
    EFI_STATUS CapsuleStatus;
} EFI_CAPSULE_RESULT_VARIABLE_HEADER;
```

**VariableTotalSize**

Size in bytes of the variable including any data beyond header as specified by CapsuleGuid.

**CapsuleGuid**

Guid from EFI_CAPSULE_HEADER
CapsuleProcessed

Timestamp using system time when processing completed.

CapsuleStatus

Result of the capsule processing. Exact interpretation of any error code may depend upon type of capsule processed.

Additional Structure When CapsuleGuid is EFI_FIRMWARE_MANAGEMENT_CAPSULE_ID_GUID

The Capsule Processing Result Variable contents always begin with

**EFI_CAPSULE_RESULT_VARIABLE_HEADER**. When CapsuleGuid is

**EFI_FIRMWARE_MANAGEMENT_CAPSULE_ID_GUID**, the header is followed by additional data as defined by **EFI_CAPSULE_RESULT_VARIABLE_FMP**.

typedef struct {
  UINT16 Version;
  UINT8 PayloadIndex;
  UINT8 UpdateImageIndex;
  EFI.Guid UpdateImageTypeId;
  // CHAR16 CapsuleFileName[];
  // CHAR16 CapsuleTarget[];
} EFI_CAPSULE_RESULT_VARIABLE_FMP;

Version

The version of this structure, currently 0x00000001.

PayloadIndex

The index, starting from zero, of the payload within the FMP capsule which was processed to generate this report.

UpdateImageIndex

The **UpdateImageIndex** from **EFI_FIRMWARE_MANAGEMENT_CAPSULE_IMAGE_HEADER** (after unsigned conversion from UINT8 to UINT16).

UpdateImageTypeId

The **UpdateImageTypeId** Guid from **EFI_FIRMWARE_MANAGEMENT_CAPSULE_IMAGE_HEADER**.

CapsuleFileName

In case of capsule loaded from disk, the zero-terminated array containing file name of capsule that was processed. In case of capsule submitted directly to **UpdateCapsule()** there is no file name, and this field is required to contain a single 16-bit zero character which is included in **VariableTotalSize**.

CapsuleTarget

This field will contain a zero-terminated CHAR16 string containing the text representation of the device path of device publishing Firmware Management Protocol (if present). In case where device path is not present and the target is not otherwise known to firmware, or when payload was blocked by policy, or skipped, this field is required to contain a single 16-bit zero character which is included in **VariableTotalSize**.
Additional Structure When CapsuleGuid is `EFI_JSON_CAPSULE_ID_GUID`

The Capsule Processing Result Variable contents always begin with `EFI_CAPSULE_RESULT_VARIABLE_HEADER`. When CapsuleGuid is `EFI_JSON_CAPSULE_ID_GUID`, the header is followed by additional data as defined by `EFI_CAPSULE_RESULT_VARIABLE_JSON`.

```c
typedef struct {
    UINT32 Version;
    UINT32 CapsuleId;
    UINT32 RespLength;
    UINT8  Resp[];
} EFI_CAPSULE_RESULT_VARIABLE_JSON;
```

**Version**
The version of this structure, currently 0x00000001.

**CapsuleId**
The unique identifier of the capsule whose processing result is recorded in this variable.

<table>
<thead>
<tr>
<th>Value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x00000000 – 0xEFFFFFFF</td>
<td>Implementation Reserved</td>
</tr>
<tr>
<td>0xF0000000 – 0xFFFFFFFF</td>
<td>Specification Reserved</td>
</tr>
</tbody>
</table>

#define REDFISH_DEFINED_JSON_SCHEMA 0xF00000000

The JSON payload shall conform to a Redfish-defined JSON schema, see DMTF-Redfish Specification.

**RespLength**
The length of `Resp` in bytes.

**Resp**
Variable length buffer containing the replied JSON payload to the caller who delivered JSON capsule to system. The definition of the JSON schema used in the replied payload is beyond the scope of this specification.
Status Codes Returned in `CapsuleStatus`

<table>
<thead>
<tr>
<th>Code</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>EFI_SUCCESS</td>
<td>Valid capsule was passed and the capsule has been successfully processed by the firmware.</td>
</tr>
<tr>
<td>EFI_INVALID_PARAMETER</td>
<td>Invalid capsule size, or an incompatible set of flags were set in the capsule header. In the case of a capsule file, the file size was not valid or an error was detected in the internal structure of the file.</td>
</tr>
<tr>
<td>EFI_DEVICE_ERROR</td>
<td>The capsule update was started, but failed due to a device error.</td>
</tr>
<tr>
<td>EFI_ACCESS_DENIED</td>
<td>Image within capsule was not loaded because the platform policy prohibits the image from being loaded.</td>
</tr>
<tr>
<td>EFI_LOAD_ERROR</td>
<td>For capsule with included driver, no driver with correct format for the platform was found.</td>
</tr>
<tr>
<td>EFI_UNSUPPORTED</td>
<td>The capsule type is not supported on this platform. Or the capsule internal structures were not recognized as valid by the platform.</td>
</tr>
<tr>
<td>EFI_OUT_OF_RESOURCES</td>
<td>There were insufficient resources to process the capsule.</td>
</tr>
<tr>
<td>EFI_NOT_READY</td>
<td>Capsule payload blocked by platform policy.</td>
</tr>
<tr>
<td>EFI_ABORTED</td>
<td>Capsule payload was skipped.</td>
</tr>
<tr>
<td>EFI_UNSUPPORTED</td>
<td>This call is not supported by this platform at the time the call is made. The platform should describe this runtime service as unsupported at runtime via an <code>EFI_RT_PROPERTIES_TABLE</code> configuration table.</td>
</tr>
</tbody>
</table>
This section defines \texttt{EFI\_LOADED\_IMAGE\_PROTOCOL} and the \texttt{EFI\_LOADED\_IMAGE\_DEVICE\_PATH\_PROTOCOL}. Respectively, these protocols describe an Image that has been loaded into memory and specifies the device path used when a PE/COFF image was loaded through the EFI Boot Service \texttt{LoadImage()}. These descriptions include the source from which the image was loaded, the current location of the image in memory, the type of memory allocated for the image, and the parameters passed to the image when it was invoked.

### 9.1 EFI Loaded Image Protocol

**EFI\_LOADED\_IMAGE\_PROTOCOL**

**Summary**

Can be used on any image handle to obtain information about the loaded image.

**GUID**

\begin{verbatim}
#define EFI_LOADED_IMAGE_PROTOCOL_GUID\ {0x5B1B31A1,0x9562,0x11d2,\}
{0x8E,0x3F,0x00,0xA0,0xC9,0x72,0x3B}\end{verbatim}

**Revision Number**

\begin{verbatim}
#define EFI_LOADED_IMAGE_PROTOCOL_REVISION 0x1000\end{verbatim}

**Protocol Interface Structure**

\begin{verbatim}
typedef struct {
    UINT32 Revision;
    EFI_HANDLE ParentHandle;
    EFI_SYSTEM_TABLE *SystemTable;

    // Source location of the image
    EFI_HANDLE DeviceHandle;
    EFI_DEVICE_PATH_PROTOCOL *FilePath;
    VOID *Reserved;

    // Image’s load options
    UINT32 LoadOptionsSize;
    VOID *LoadOptions;

    // Location where image was loaded
    VOID *ImageBase;
    UINT64 ImageSize;
    EFI_MEMORY_TYPE ImageCodeType;
    EFI_MEMORY_TYPE ImageDataType;
    EFI_IMAGE_UNLOAD Unload;
} EFI_LOADED_IMAGE_PROTOCOL;
\end{verbatim}
Parameters

Revision
Defines the revision of the EFI_LOADED_IMAGE_PROTOCOL structure. All future revisions will be backward compatible to the current revision.

ParentHandle
Parent image's image handle. NULL if the image is loaded directly from the firmware's boot manager. Type EFI_HANDLE is defined in Section 7.

SystemTable
The image's EFI system table pointer. Type EFI_SYSTEM_TABLE is defined in Section 4.

DeviceHandle
The device handle that the EFI Image was loaded from. Type EFI_HANDLE is defined in Section 7.

FilePath
A pointer to the file path portion specific to DeviceHandle that the EFI Image was loaded from. EFI_DEVICE_PATH_PROTOCOL is defined in Section 10.2.

Reserved
Reserved. DO NOT USE.

LoadOptionsSize
The size in bytes of LoadOptions.

LoadOptions
A pointer to the image's binary load options. See the OptionalData parameter in the 1 section of the Boot Manager chapter for information on the source of the LoadOptions data.

ImageBase
The base address at which the image was loaded.

ImageSize
The size in bytes of the loaded image.

ImageCodeType
The memory type that the code sections were loaded as. Type EFI_MEMORY_TYPE is defined in Section 7.

ImageDataType
The memory type that the data sections were loaded as. Type EFI_MEMORY_TYPE is defined in Section 7.

Unload
Function that unloads the image. See Unload().

Description
Each loaded image has an image handle that supports EFI_LOADED_IMAGE_PROTOCOL. When an image is started, it is passed the image handle for itself. The image can use the handle to obtain its relevant image data stored in the EFI_LOADED_IMAGE_PROTOCOL structure, such as its load options.

EFI_LOADED_IMAGE_PROTOCOLUnload()

Summary
Unloads an image from memory.

Prototype
typedef  
  EFI_STATUS  
  (EFIAPI *EFI_IMAGE_UNLOAD) (  
    IN EFI_HANDLE  ImageHandle,  
  );

Parameters
ImageHandle  The handle to the image to unload. Type EFI_HANDLE is defined in Section.

Description
The Unload() function is a callback that a driver registers to do cleanup when the UnloadImage boot service function is called.

Status Codes Returned
<table>
<thead>
<tr>
<th>Code</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>EFI_SUCCESS</td>
<td>The image was unloaded.</td>
</tr>
<tr>
<td>EFI_INVALID_PARAMETER</td>
<td>The ImageHandle was not valid.</td>
</tr>
</tbody>
</table>

9.2 EFI Loaded Image Device Path Protocol

EFI_LOADED_IMAGE_DEVICE_PATH_PROTOCOL

Summary
When installed, the Loaded Image Device Path Protocol specifies the device path that was used when a PE/COFF image was loaded through the EFI Boot Service LoadImage().

GUID
#define EFI_LOADED_IMAGE_DEVICE_PATH_PROTOCOL_GUID \  
  {0xbc62157e,0x3e33,0x4fec,\  
  {0x99,0x20,0x2d,0x3b,0x36,0xd7,0xdf}}

Description
The Loaded Image Device Path Protocol uses the same protocol interface structure as the Device Path Protocol defined in Chapter 9. The only difference between the Device Path Protocol and the Loaded Image Device Path Protocol is the protocol GUID value.

The Loaded Image Device Path Protocol must be installed onto the image handle of a PE/COFF image loaded through the EFI Boot Service LoadImage(). A copy of the device path specified by the DevicePath parameter to the EFI Boot Service LoadImage() is made before it is installed onto the image handle. It is legal to call LoadImage() for a buffer in memory with a NULL DevicePath parameter. In this case, the Loaded Image Device Path Protocol is installed with a NULL interface pointer.
This section contains the definition of the device path protocol and the information needed to construct and manage device paths in the UEFI environment. A device path is constructed and used by the firmware to convey the location of important devices, such as the boot device and console, consistent with the software-visible topology of the system.

### 10.1 Device Path Overview

A Device Path is used to define the programmatic path to a device. The primary purpose of a Device Path is to allow an application, such as an OS loader, to determine the physical device that the interfaces are abstracting.

A collection of device paths is usually referred to as a name space. ACPI, for example, is rooted around a name space that is written in ASL (ACPI Source Language). Given that EFI does not replace ACPI and defers to ACPI when ever possible, it would seem logical to utilize the ACPI name space in EFI. However, the ACPI name space was designed for usage at operating system runtime and does not fit well in platform firmware or OS loaders. Given this, EFI defines its own name space, called a Device Path.

A Device Path is designed to make maximum leverage of the ACPI name space. One of the key structures in the Device Path defines the linkage back to the ACPI name space. The Device Path also is used to fill in the gaps where ACPI defers to buses with standard enumeration algorithms. The Device Path is able to relate information about which device is being used on buses with standard enumeration mechanisms. The Device Path is also used to define the location on a medium where a file should be, or where it was loaded from. A special case of the Device Path can also be used to support the optional booting of legacy operating systems from legacy media.

The Device Path was designed so that the OS loader and the operating system could tell which devices the platform firmware was using as boot devices. This allows the operating system to maintain a view of the system that is consistent with the platform firmware. An example of this is a “headless” system that is using a network connection as the boot device and console. In such a case, the firmware will convey to the operating system the network adapter and network protocol information being used as the console and boot device in the device path for these devices.

### 10.2 EFI Device Path Protocol

This section provides a detailed description of `EFI_DEVICE_PATH_PROTOCOL`.

**EFI_DEVICE_PATH_PROTOCOL**

**Summary**

Can be used on any device handle to obtain generic path/location information concerning the physical device or logical device. If the handle does not logically map to a physical device, the handle may not necessarily support the device path protocol. The device path describes the location of the device the handle is for. The size of the Device Path can be determined from the structures that make up the Device Path.
GUID

```c
#define EFI_DEVICE_PATH_PROTOCOL_GUID "\0x09576e91,0x6d3f,0x11d2,\0x8e,0x39,0x00,0xa0,0xc9,0x69,0x72,0x3b"
```

Protocol Interface Structure

```c
//*******************************************************
// EFI_DEVICE_PATH_PROTOCOL
//*******************************************************
typedef struct _EFI_DEVICE_PATH_PROTOCOL {
    UINT8 Type;
    UINT8 SubType;
    UINT8 Length[2];
} EFI_DEVICE_PATH_PROTOCOL;
```

Description

The executing UEFI Image may use the device path to match its own device drivers to the particular device. Note that the executing UEFI OS loader and UEFI application images must access all physical devices via Boot Services device handles until `EFI_BOOT_SERVICES.ExitBootServices()` is successfully called. A UEFI driver may access only a physical device for which it provides functionality.

10.3 Device Path Nodes

There are six major types of Device Path nodes:

- **Hardware Device Path.** This Device Path defines how a device is attached to the resource domain of a system, where resource domain is simply the shared memory, memory mapped I/O, and I/O space of the system.

- **ACPI Device Path.** This Device Path is used to describe devices whose enumeration is not described in an industry-standard fashion. These devices must be described using ACPI AML in the ACPI name space; this Device Path is a linkage to the ACPI name space.

- **Messaging Device Path.** This Device Path is used to describe the connection of devices outside the resource domain of the system. This Device Path can describe physical messaging information such as a SCSI ID, or abstract information such as networking protocol IP addresses.

- **Media Device Path.** This Device Path is used to describe the portion of a medium that is being abstracted by a boot service. For example, a Media Device Path could define which partition on a hard drive was being used.

- **BIOS Boot Specification Device Path.** This Device Path is used to point to boot legacy operating systems; it is based on the BIOS Boot Specification Version 1.01. Refer to Appendix Q for details on obtaining this specification.

- **End of Hardware Device Path.** Depending on the Sub-Type, this Device Path node is used to indicate the end of the Device Path instance or Device Path structure.
10.3.1 Generic Device Path Structures

A Device Path is a variable-length binary structure that is made up of variable-length generic Device Path nodes. Table 10-1 defines the structure of a variable-length generic Device Path node and the lengths of its components. The table defines the type and sub-type values corresponding to the Device Paths described in Section 10.3; all other type and sub-type values are Reserved.

Table 10-1 Generic Device Path Node Structure

<table>
<thead>
<tr>
<th>Mnemonic</th>
<th>Byte Offset</th>
<th>Byte Length</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Type</td>
<td>0</td>
<td>1</td>
<td>Type 0x01 – Hardware Device Path</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>Type 0x02 – ACPI Device Path</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>Type 0x03 – Messaging Device Path</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>Type 0x04 – Media Device Path</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>Type 0x05 – BIOS Boot Specification Device Path</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>Type 0x7F – End of Hardware Device Path</td>
</tr>
<tr>
<td>Sub-Type</td>
<td>1</td>
<td>1</td>
<td>Sub-Type – Varies by Type. (See Table 10-2.)</td>
</tr>
<tr>
<td>Length</td>
<td>2</td>
<td>2</td>
<td>Length of this structure in bytes. Length is 4 + n bytes.</td>
</tr>
<tr>
<td>Specific Device Path Data</td>
<td>4</td>
<td>n</td>
<td>Specific Device Path data. Type and Sub-Type define type of data. Size of data is included in Length.</td>
</tr>
</tbody>
</table>

A Device Path is a series of generic Device Path nodes. The first Device Path node starts at byte offset zero of the Device Path. The next Device Path node starts at the end of the previous Device Path node. Therefore all nodes are byte-packed data structures that may appear on any byte boundary. All code references to device path notes must assume all fields are unaligned. Since every Device Path node contains a length field in a known place, it is possible to traverse Device Path nodes that are of an unknown type. There is no limit to the number, type, or sequence of nodes in a Device Path.

A Device Path is terminated by an End of Hardware Device Path node. This type of node has two sub-types (see Table 10-2):

- **End This Instance of a Device Path** (sub-type 0x01). This type of node terminates one Device Path instance and denotes the start of another. This is only required when an environment variable represents multiple devices. An example of this would be the **ConsoleOut** environment variable that consists of both a VGA console and serial output console. This variable would describe a console output stream that is sent to both VGA and serial concurrently and thus has a Device Path that contains two complete Device Paths.

- **End Entire Device Path** (sub-type 0xFF). This type of node terminates an entire Device Path. Software searches for this sub-type to find the end of a Device Path. All Device Paths must end with this sub-type.

Table 10-2 Device Path End Structure

<table>
<thead>
<tr>
<th>Mnemonic</th>
<th>Byte Offset</th>
<th>Byte Length</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Type</td>
<td>0</td>
<td>1</td>
<td>Type 0x7F – End of Hardware Device Path</td>
</tr>
</tbody>
</table>
10.3.2 Hardware Device Path

This Device Path defines how a device is attached to the resource domain of a system, where resource domain is simply the shared memory, memory mapped I/O, and I/O space of the system. It is possible to have multiple levels of Hardware Device Path such as a PCCARD device that was attached to a PCCARD PCI controller.

10.3.2.1 PCI Device Path

The Device Path for PCI defines the path to the PCI configuration space address for a PCI device. There is one PCI Device Path entry for each device and function number that defines the path from the root PCI bus to the device. Because the PCI bus number of a device may potentially change, a flat encoding of single PCI Device Path entry cannot be used. An example of this is when a PCI device is behind a bridge, and one of the following events occurs:

- OS performs a Plug and Play configuration of the PCI bus.
- A hot plug of a PCI device is performed.
- The system configuration changes between reboots.

The PCI Device Path entry must be preceded by an ACPI Device Path entry that uniquely identifies the PCI root bus. The programming of root PCI bridges is not defined by any PCI specification and this is why an ACPI Device Path entry is required.

Table 10-3 PCI Device Path

<table>
<thead>
<tr>
<th>Mnemonic</th>
<th>Byte Offset</th>
<th>Byte Length</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Type</td>
<td>0</td>
<td>1</td>
<td>Type 1 – Hardware Device Path</td>
</tr>
<tr>
<td>Sub-Type</td>
<td>1</td>
<td>1</td>
<td>Sub-Type 1 – PCI</td>
</tr>
<tr>
<td>Length</td>
<td>2</td>
<td>2</td>
<td>Length of this structure is 6 bytes</td>
</tr>
<tr>
<td>Function</td>
<td>4</td>
<td>1</td>
<td>PCI Function Number</td>
</tr>
<tr>
<td>Device</td>
<td>5</td>
<td>1</td>
<td>PCI Device Number</td>
</tr>
</tbody>
</table>

10.3.2.2 PCCARD Device Path

Table 10-4 PCCARD Device Path

<table>
<thead>
<tr>
<th>Mnemonic</th>
<th>Byte Offset</th>
<th>Byte Length</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Type</td>
<td>0</td>
<td>1</td>
<td>Type 1 – Hardware Device Path</td>
</tr>
<tr>
<td>Sub-Type</td>
<td>1</td>
<td>1</td>
<td>Sub-Type 2 – PCCARD</td>
</tr>
<tr>
<td>Length</td>
<td>2</td>
<td>2</td>
<td>Length of this structure in bytes. Length is 5 bytes.</td>
</tr>
</tbody>
</table>
10.3.2.3 Memory Mapped Device Path

Table 10-5 Memory Mapped Device Path

<table>
<thead>
<tr>
<th>Mnemonic</th>
<th>Byte Offset</th>
<th>Byte Length</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Type</td>
<td>0</td>
<td>1</td>
<td>Type 1 – Hardware Device Path.</td>
</tr>
<tr>
<td>Sub-Type</td>
<td>1</td>
<td>1</td>
<td>Sub-Type 3 – Memory Mapped.</td>
</tr>
<tr>
<td>Length</td>
<td>2</td>
<td>2</td>
<td>Length of this structure in bytes. Length is 24 bytes.</td>
</tr>
<tr>
<td>Memory Type</td>
<td>4</td>
<td>4</td>
<td>Type EFI_MEMORY_TYPE. Type EFI_MEMORY_TYPE is defined in the EFI_BOOT_SERVICES.AllocatePages() function description.</td>
</tr>
<tr>
<td>Start Address</td>
<td>8</td>
<td>8</td>
<td>Starting Memory Address.</td>
</tr>
<tr>
<td>End Address</td>
<td>16</td>
<td>8</td>
<td>Ending Memory Address.</td>
</tr>
</tbody>
</table>

10.3.2.4 Vendor Device Path

The Vendor Device Path allows the creation of vendor-defined Device Paths. A vendor must allocate a Vendor GUID for a Device Path. The Vendor GUID can then be used to define the contents on the $n$ bytes that follow in the Vendor Device Path node.

Table 10-6 Vendor-Defined Device Path

<table>
<thead>
<tr>
<th>Mnemonic</th>
<th>Byte Offset</th>
<th>Byte Length</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Type</td>
<td>0</td>
<td>1</td>
<td>Type 1 – Hardware Device Path.</td>
</tr>
<tr>
<td>Sub-Type</td>
<td>1</td>
<td>1</td>
<td>Sub-Type 4 – Vendor.</td>
</tr>
<tr>
<td>Length</td>
<td>2</td>
<td>2</td>
<td>Length of this structure in bytes. Length is $20 + n$ bytes.</td>
</tr>
<tr>
<td>Vendor_GUID</td>
<td>4</td>
<td>16</td>
<td>Vendor-assigned GUID that defines the data that follows.</td>
</tr>
<tr>
<td>Vendor Defined Data</td>
<td>20</td>
<td>$n$</td>
<td>Vendor-defined variable size data.</td>
</tr>
</tbody>
</table>

10.3.2.5 Controller Device Path

Table 10-7 Controller Device Path

<table>
<thead>
<tr>
<th>Mnemonic</th>
<th>Byte Offset</th>
<th>Byte Length</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Type</td>
<td>0</td>
<td>1</td>
<td>Type 1 – Hardware Device Path.</td>
</tr>
<tr>
<td>Sub-Type</td>
<td>1</td>
<td>1</td>
<td>Sub-Type 5 – Controller.</td>
</tr>
<tr>
<td>Length</td>
<td>2</td>
<td>2</td>
<td>Length of this structure in bytes. Length is 8 bytes.</td>
</tr>
<tr>
<td>Controller Number</td>
<td>4</td>
<td>4</td>
<td>Controller number.</td>
</tr>
</tbody>
</table>
10.3.2.6 BMC Device Path

The Device Path for a Baseboard Management Controller (BMC) host interface.

Table 10-8 BMC Device Path

<table>
<thead>
<tr>
<th>Mnemonic</th>
<th>Byte Offset</th>
<th>Byte Length</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Type</td>
<td>0</td>
<td>1</td>
<td>Type 1 – Hardware Device Path.</td>
</tr>
<tr>
<td>Sub-Type</td>
<td>1</td>
<td>1</td>
<td>Sub Type 6 – BMC</td>
</tr>
<tr>
<td>Length</td>
<td>2</td>
<td>2</td>
<td>Length of this structure in bytes. Length is 13 bytes.</td>
</tr>
<tr>
<td>Interface Type</td>
<td>4</td>
<td>1</td>
<td>The Baseboard Management Controller (BMC) host interface type:</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>0x00: Unknown</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>0x01: KCS: Keyboard Controller Style</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>0x02: SMIC: Server Management Interface Chip</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>0x03: BT: Block Transfer</td>
</tr>
<tr>
<td>Base Address</td>
<td>5</td>
<td>8</td>
<td>Base address (either memory-mapped or I/O) of the BMC.</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>If the least-significant bit of the field is a 1, the address is in I/O</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>space; otherwise, the address is memory-mapped. Refer to the IPMI</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>Interface Specification for usage details.</td>
</tr>
</tbody>
</table>

10.3.3 ACPI Device Path

This Device Path contains ACPI Device IDs that represent a device’s Plug and Play Hardware ID and its corresponding unique persistent ID. The ACPI IDs are stored in the ACPI _HID, _CID, and _UID device identification objects that are associated with a device. The ACPI Device Path contains values that must match exactly the ACPI name space that is provided by the platform firmware to the operating system. Refer to the ACPI specification for a complete description of the _HID, _CID, and _UID device identification objects.

The _HID and _CID values are optional device identification objects that appear in the ACPI name space. If only _HID is present, the _HID must be used to describe any device that will be enumerated by the ACPI driver. The _CID, if present, contains information that is important for the OS to attach generic driver (e.g., PCI Bus Driver), while the _HID contains information important for the OS to attach device-specific driver. The ACPI bus driver only enumerates a device when no standard bus enumerator exists for a device.

The _UID object provides the OS with a serial number-style ID for a device that does not change across reboots. The object is optional, but is required when a system contains two devices that report the same _HID. The _UID only needs to be unique among all device objects with the same _HID value. If no _UID exists in the ACPI name space for a _HID the value of zero must be stored in the _UID field of the ACPI Device Path.

The ACPI Device Path is only used to describe devices that are not defined by a Hardware Device Path. An _HID (along with _CID if present) is required to represent a PCI root bridge, since the PCI specification does not define the programming model for a PCI root bridge. There are two subtypes of the ACPI Device Path: a simple subtype that only includes the _HID and _UID fields, and an extended subtype that includes the _HID, _CID, and _UID fields.
The ACPI Device Path node only supports numeric 32-bit values for the _HID and _UID values. The Expanded ACPI Device Path node supports both numeric and string values for the _HID, _UID, and _CID values. As a result, the ACPI Device Path node is smaller and should be used if possible to reduce the size of device paths that may potentially be stored in nonvolatile storage. If a string value is required for the _HID field, or a string value is required for the _UID field, or a _CID field is required, then the Expanded ACPI Device Path node must be used. If a string field of the Expanded ACPI Device Path node is present, then the corresponding numeric field is ignored.

The _HID and _CID fields in the ACPI Device Path node and Expanded ACPI Device Path node are stored as a 32-bit compressed EISA-type IDs. The following macro can be used to compute these EISA-type IDs from a Plug and Play Hardware ID. The Plug and Play Hardware IDs used to compute the _HID and _CID fields in the EFI device path nodes must match the Plug and Play Hardware IDs used to build the matching entries in the ACPI tables. The compressed EISA-type IDs produced by this macro differ from the compressed EISA-type IDs stored in ACPI tables. As a result, the compressed EISA-type IDs from the ACPI Device Path nodes cannot be directly compared to the compressed EISA-type IDs from the ACPI table.

```
#define EFI_PNP_ID(ID) (UINT32)(((ID) << 16) | 0x41D0)
#define EISA_PNP_ID(ID) EFI_PNP_ID(ID)
```

**Table 10-9 ACPI Device Path**

<table>
<thead>
<tr>
<th>Mnemonic</th>
<th>Byte Offset</th>
<th>Byte Length</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Type</td>
<td>0</td>
<td>1</td>
<td>Type 2 – ACPI Device Path.</td>
</tr>
<tr>
<td>Sub-Type</td>
<td>1</td>
<td>1</td>
<td>Sub-Type 1 ACPI Device Path.</td>
</tr>
<tr>
<td>Length</td>
<td>2</td>
<td>2</td>
<td>Length of this structure in bytes. Length is 12 bytes.</td>
</tr>
<tr>
<td>_HID</td>
<td>4</td>
<td>4</td>
<td>Device’s PnP hardware ID stored in a numeric 32-bit compressed EISA-type ID. This value must match the corresponding _HID in the ACPI name space.</td>
</tr>
<tr>
<td>_UID</td>
<td>8</td>
<td>4</td>
<td>Unique ID that is required by ACPI if two devices have the same _HID. This value must also match the corresponding _UID/_HID pair in the ACPI name space. Only the 32-bit numeric value type of _UID is supported; thus strings must not be used for the _UID in the ACPI name space.</td>
</tr>
</tbody>
</table>

**Table 10-10 Expanded ACPI Device Path**

<table>
<thead>
<tr>
<th>Mnemonic</th>
<th>Byte Offset</th>
<th>Byte Length</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Type</td>
<td>0</td>
<td>1</td>
<td>Type 2 – ACPI Device Path.</td>
</tr>
<tr>
<td>Sub-Type</td>
<td>1</td>
<td>1</td>
<td>Sub-Type 2 Expanded ACPI Device Path.</td>
</tr>
<tr>
<td>Length</td>
<td>2</td>
<td>2</td>
<td>Length of this structure in bytes. Minimum length is 19 bytes. The actual size will depend on the size of the _HIDSTR, _UIDSTR, and _CIDSTR fields.</td>
</tr>
<tr>
<td>_HID</td>
<td>4</td>
<td>4</td>
<td>Device’s PnP hardware ID stored in a numeric 32-bit compressed EISA-type ID. This value must match the corresponding _HID in the ACPI name space.</td>
</tr>
</tbody>
</table>
10.3.3.1 ACPI _ADR Device Path

The _ADR device path is used to contain video output device attributes to support the Graphics Output Protocol. The device path can contain multiple _ADR entries if multiple video output devices are displaying the same output.

Table 10-11 ACPI _ADR Device Path

<table>
<thead>
<tr>
<th>Mnemonic</th>
<th>Byte Offset</th>
<th>Byte Length</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Type</td>
<td>0</td>
<td>1</td>
<td>Type 2 – ACPI Device Path</td>
</tr>
<tr>
<td>Sub-Type</td>
<td>1</td>
<td>1</td>
<td>Sub-Type3 _ADR Device Path</td>
</tr>
<tr>
<td>Length</td>
<td>2</td>
<td>2</td>
<td>Length of this structure in bytes. Minimum length is 8.</td>
</tr>
<tr>
<td>_ADR</td>
<td>4</td>
<td>4</td>
<td>_ADR value. For video output devices the value of this field comes from Table B-2 ACPI 3.0 specification. At least one _ADR value is required</td>
</tr>
<tr>
<td>Additional _ADR</td>
<td>8</td>
<td>N</td>
<td>This device path may optionally contain more than one _ADR entry.</td>
</tr>
</tbody>
</table>
10.3.3.2 NVDIMM Device Path
This device path describes an NVDIMM device using the ACPI 6.0 specification defined NFIT Device Handle as the identifier.

Table 10-12 NVDIMM Device Path

<table>
<thead>
<tr>
<th>Mnemonic</th>
<th>Byte Offset</th>
<th>Byte Length</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Type</td>
<td>0</td>
<td>1</td>
<td>Type 2 - ACPI Device Path</td>
</tr>
<tr>
<td>Sub-Type</td>
<td>1</td>
<td>1</td>
<td>Sub-type 4 - NVDIMM Device</td>
</tr>
<tr>
<td>Length</td>
<td>2</td>
<td>2</td>
<td>8 – Single NFIT Device Handle is supported.</td>
</tr>
<tr>
<td>NFIT Device</td>
<td>4</td>
<td>4</td>
<td>NFIT Device Handle - Unique physical identifier. See ACPI Defined Devices and Device Specific Objects section, NVDIMM Devices sub-chapter for the specific definition of the fields utilized for this handle.</td>
</tr>
</tbody>
</table>

10.3.4 Messaging Device Path
This Device Path is used to describe the connection of devices outside the resource domain of the system. This Device Path can describe physical messaging information like SCSI ID, or abstract information like networking protocol IP addresses.

10.3.4.1 ATAPI Device Path

Table 10-13 ATAPI Device Path

<table>
<thead>
<tr>
<th>Mnemonic</th>
<th>Byte Offset</th>
<th>Byte Length</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Type</td>
<td>0</td>
<td>1</td>
<td>Type 3 – Messaging Device Path</td>
</tr>
<tr>
<td>Sub-Type</td>
<td>1</td>
<td>1</td>
<td>Sub-Type 1 – ATAPI</td>
</tr>
<tr>
<td>Length</td>
<td>2</td>
<td>2</td>
<td>Length of this structure in bytes. Length is 8 bytes.</td>
</tr>
<tr>
<td>PrimarySecondary</td>
<td>4</td>
<td>1</td>
<td>Set to zero for primary or one for secondary</td>
</tr>
<tr>
<td>SlaveMaster</td>
<td>5</td>
<td>1</td>
<td>Set to zero for master or one for slave mode</td>
</tr>
<tr>
<td>Logical Unit Number</td>
<td>6</td>
<td>2</td>
<td>Logical Unit Number</td>
</tr>
</tbody>
</table>
10.3.4.2 SCSI Device Path

Table 10-14 SCSI Device Path

<table>
<thead>
<tr>
<th>Mnemonic</th>
<th>Byte Offset</th>
<th>Byte Length</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Type</td>
<td>0</td>
<td>1</td>
<td>Type 3 – Messaging Device Path</td>
</tr>
<tr>
<td>Sub-Type</td>
<td>1</td>
<td>1</td>
<td>Sub-Type 2 – SCSI</td>
</tr>
<tr>
<td>Length</td>
<td>2</td>
<td>2</td>
<td>Length of this structure in bytes. Length is 8 bytes.</td>
</tr>
<tr>
<td>Target ID</td>
<td>4</td>
<td>2</td>
<td>Target ID on the SCSI bus (PUN)</td>
</tr>
<tr>
<td>Logical Unit Number</td>
<td>6</td>
<td>2</td>
<td>Logical Unit Number (LUN)</td>
</tr>
</tbody>
</table>

10.3.4.3 Fibre Channel Device Path

Table 10-15 Fibre Channel Device Path

<table>
<thead>
<tr>
<th>Mnemonic</th>
<th>Byte Offset</th>
<th>Byte Length</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Type</td>
<td>0</td>
<td>1</td>
<td>Type 3 – Messaging Device Path</td>
</tr>
<tr>
<td>Sub-Type</td>
<td>1</td>
<td>1</td>
<td>Sub-Type 3 – Fibre Channel</td>
</tr>
<tr>
<td>Length</td>
<td>2</td>
<td>2</td>
<td>Length of this structure in bytes. Length is 24 bytes.</td>
</tr>
<tr>
<td>Reserved</td>
<td>4</td>
<td>4</td>
<td>Reserved</td>
</tr>
<tr>
<td>World Wide Name</td>
<td>8</td>
<td>8</td>
<td>Fibre Channel World Wide Name</td>
</tr>
<tr>
<td>Logical Unit Number</td>
<td>16</td>
<td>8</td>
<td>Fibre Channel Logical Unit Number</td>
</tr>
</tbody>
</table>

Table 10-16 Fibre Channel Ex Device Path

<table>
<thead>
<tr>
<th>Mnemonic</th>
<th>Byte Offset</th>
<th>Byte Length</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Type</td>
<td>0</td>
<td>1</td>
<td>Type 3 – Messaging Device Path</td>
</tr>
<tr>
<td>Sub-Type</td>
<td>1</td>
<td>1</td>
<td>Sub-Type 21 – Fibre Channel Ex</td>
</tr>
<tr>
<td>Length</td>
<td>2</td>
<td>2</td>
<td>Length of this structure in bytes. Length is 24 bytes.</td>
</tr>
<tr>
<td>Reserved</td>
<td>4</td>
<td>4</td>
<td>Reserved</td>
</tr>
<tr>
<td>World Wide Name</td>
<td>8</td>
<td>8</td>
<td>8 byte array containing Fibre Channel End Device Port Name (a.k.a., World Wide Name)</td>
</tr>
<tr>
<td>Logical Unit Number</td>
<td>16</td>
<td>8</td>
<td>8 byte array containing Fibre Channel Logical Unit Number</td>
</tr>
</tbody>
</table>

The Fibre Channel Ex device path clarifies the definition of the Logical Unit Number field to conform with the T-10 SCSI Architecture Model 4 specification. The 8 byte Logical Unit Number field in the device path must conform with a logical unit number returned by a SCSI REPORT LUNS command.

When the Fibre Channel Ex Device Path is used with the Extended SCSI Pass Thru Protocol the UINT64 LUN argument must be converted to the eight byte array Logical Unit Number field in the device path by
treat the eight byte array as an EFI UINT64. For example a Logical Unit Number array of \{0,1,2,3,4,5,6,7\} becomes a UINT64 of 0x0706050403020100.

When an application client displays or otherwise makes a 64-bit LUN visible to a user, it should be done in conformance with SAM-4. SAM-4 requires a LUN to be displayed in hexadecimal format with byte 0 first (i.e., on the left) and byte 7 last (i.e., on the right) regardless of the internal representation of the LUN. UEFI defines all data structures a “little endian” and SCSI defines all data structures as “big endian”. Fibre Channel Ex Device Path Example shows an example device path for a Fibre Channel controller on a typical UEFI platform. This Fibre Channel Controller is connected to the port 0 of the root hub, and its interface number is 0. The Fibre Channel Host Controller is a PCI device whose PCI device number 0x1F and PCI function 0x00. So, the whole device path for this Fibre Channel Controller consists an ACPI Device Path Node, a PCI Device Path Node, a Fibre Channel Device Path Node and a Device Path End Structure. The _HID and _UID must match the ACPI table description of the PCI Root Bridge. The Fibre Channel WWN and LUN were picked to show byte order and they are not typical real world values. The shorthand notation for this device path is:

\texttt{PciRoot(0)/PCI(31,0)/FibreEx(0x0001020304050607, 0x0001020304050607)}

<table>
<thead>
<tr>
<th>Byte Offset</th>
<th>Byte Length</th>
<th>Data</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>1</td>
<td>0x02</td>
<td>Generic Device Path Header – Type ACPI Device Path</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>0x01</td>
<td>Sub type – ACPI Device Path</td>
</tr>
<tr>
<td>2</td>
<td>2</td>
<td>0x0C</td>
<td>Length – 0x0C bytes</td>
</tr>
<tr>
<td>4</td>
<td>4</td>
<td>0x41D0, 0x0A03</td>
<td>_HID PNP0A03 – 0x41D0 represents the compressed string ‘PNP’ and is encoded in the low order bytes. The compression method is described in the ACPI Specification.</td>
</tr>
<tr>
<td>8</td>
<td>4</td>
<td>0x0000</td>
<td>_UID</td>
</tr>
<tr>
<td>12</td>
<td>1</td>
<td>0x01</td>
<td>Generic Device Path Header – Type Hardware Device Path</td>
</tr>
<tr>
<td>13</td>
<td>1</td>
<td>0x01</td>
<td>Sub type – PCI</td>
</tr>
<tr>
<td>14</td>
<td>2</td>
<td>0x06</td>
<td>Length – 0x06 bytes</td>
</tr>
<tr>
<td>16</td>
<td>1</td>
<td>0x0</td>
<td>PCI Function</td>
</tr>
<tr>
<td>17</td>
<td>1</td>
<td>0x1F</td>
<td>PCI Device</td>
</tr>
<tr>
<td>18</td>
<td>1</td>
<td>0x03</td>
<td>Generic Device Path Header – Type Message Device Path</td>
</tr>
<tr>
<td>19</td>
<td>1</td>
<td>0x15</td>
<td>Sub type – Fibre Channel Ex</td>
</tr>
<tr>
<td>20</td>
<td>2</td>
<td>0x14</td>
<td>Length – 20 bytes</td>
</tr>
<tr>
<td>21</td>
<td>1</td>
<td>0x00</td>
<td>8 byte array containing Fibre Channel End Device Port Name (a.k.a., World Wide Name)</td>
</tr>
<tr>
<td>22</td>
<td>1</td>
<td>0x01</td>
<td></td>
</tr>
<tr>
<td>23</td>
<td>1</td>
<td>0x02</td>
<td></td>
</tr>
<tr>
<td>24</td>
<td>1</td>
<td>0x03</td>
<td></td>
</tr>
<tr>
<td>25</td>
<td>1</td>
<td>0x04</td>
<td></td>
</tr>
<tr>
<td>26</td>
<td>1</td>
<td>0x05</td>
<td></td>
</tr>
<tr>
<td>27</td>
<td>1</td>
<td>0x06</td>
<td></td>
</tr>
</tbody>
</table>
### 10.3.4.4 1394 Device Path

**Table 10-18 1394 Device Path**

<table>
<thead>
<tr>
<th>Mnemonic</th>
<th>Byte Offset</th>
<th>Byte Length</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Type</td>
<td>0</td>
<td>1</td>
<td>Type 3 – Messaging Device Path</td>
</tr>
<tr>
<td>Sub-Type</td>
<td>1</td>
<td>1</td>
<td>Sub-Type 4 – 1394</td>
</tr>
<tr>
<td>Length</td>
<td>2</td>
<td>2</td>
<td>Length of this structure in bytes. Length is 16 bytes.</td>
</tr>
<tr>
<td>Reserved</td>
<td>4</td>
<td>4</td>
<td>Reserved</td>
</tr>
<tr>
<td>GUID(^1)</td>
<td>8</td>
<td>8</td>
<td>1394 Global Unique ID (GUID)(^1)</td>
</tr>
</tbody>
</table>

Note: 1 The usage of the term GUID is per the 1394 specification. This is not the same as the EFI_GUID type defined in the EFI Specification.

### 10.3.4.5 USB Device Paths

**Table 10-19 USB Device Path**

<table>
<thead>
<tr>
<th>Mnemonic</th>
<th>Byte Offset</th>
<th>Byte Length</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Type</td>
<td>0</td>
<td>1</td>
<td>Type 3 – Messaging Device Path</td>
</tr>
<tr>
<td>Sub-Type</td>
<td>1</td>
<td>1</td>
<td>Sub-Type 5 – USB</td>
</tr>
<tr>
<td>Length</td>
<td>2</td>
<td>2</td>
<td>Length of this structure in bytes. Length is 6 bytes.</td>
</tr>
<tr>
<td>USB Parent Port Number</td>
<td>4</td>
<td>1</td>
<td>USB Parent Port Number</td>
</tr>
<tr>
<td>Interface</td>
<td>5</td>
<td>1</td>
<td>USB Interface Number</td>
</tr>
</tbody>
</table>

### 10.3.4.5.1 USB Device Path Example

**Table 10-20** shows an example device path for a USB controller on a desktop platform. This USB Controller is connected to the port 0 of the root hub, and its interface number is 0. The USB Host
Controller is a PCI device whose PCI device number 0x1F and PCI function 0x02. So, the whole device path for this USB Controller consists an ACPI Device Path Node, a PCI Device Path Node, a USB Device Path Node and a Device Path End Structure. The _HID and _UID must match the ACPI table description of the PCI Root Bridge. The shorthand notation for this device path is:

\[
\text{PciRoot(0)/PCI(31,2)/USB(0,0)}.
\]

**Table 10-20 USB Device Path Examples**

<table>
<thead>
<tr>
<th>Byte Offset</th>
<th>Byte Length</th>
<th>Data</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x00</td>
<td>0x01</td>
<td>0x02</td>
<td><strong>Generic Device Path Header</strong> – Type ACPI Device Path</td>
</tr>
<tr>
<td>0x01</td>
<td>0x01</td>
<td>0x01</td>
<td>Sub type – ACPI Device Path</td>
</tr>
<tr>
<td>0x02</td>
<td>0x02</td>
<td>0x0C</td>
<td>Length – 0x0C bytes</td>
</tr>
<tr>
<td>0x04</td>
<td>0x04</td>
<td>0x41D0, 0x0A03</td>
<td>_HID PNP0A03 – 0x41D0 represents the compressed string ‘PNP’ and is encoded in the low order bytes. The compression method is described in the ACPI Specification.</td>
</tr>
<tr>
<td>0x08</td>
<td>0x04</td>
<td>0x0000</td>
<td>_UID</td>
</tr>
<tr>
<td>0x0C</td>
<td>0x01</td>
<td>0x01</td>
<td><strong>Generic Device Path Header</strong> – Type Hardware Device Path</td>
</tr>
<tr>
<td>0x0D</td>
<td>0x01</td>
<td>0x01</td>
<td>Sub type – PCI</td>
</tr>
<tr>
<td>0x0E</td>
<td>0x02</td>
<td>0x06</td>
<td>Length – 0x06 bytes</td>
</tr>
<tr>
<td>0x10</td>
<td>0x01</td>
<td>0x02</td>
<td>PCI Function</td>
</tr>
<tr>
<td>0x11</td>
<td>0x01</td>
<td>0x1F</td>
<td>PCI Device</td>
</tr>
<tr>
<td>0x12</td>
<td>0x01</td>
<td>0x03</td>
<td><strong>Generic Device Path Header</strong> – Type Message Device Path</td>
</tr>
<tr>
<td>0x13</td>
<td>0x01</td>
<td>0x05</td>
<td>Sub type – USB</td>
</tr>
<tr>
<td>0x14</td>
<td>0x02</td>
<td>0x06</td>
<td>Length – 0x06 bytes</td>
</tr>
<tr>
<td>0x16</td>
<td>0x01</td>
<td>0x00</td>
<td>Parent Hub Port Number</td>
</tr>
<tr>
<td>0x17</td>
<td>0x01</td>
<td>0x00</td>
<td>Controller Interface Number</td>
</tr>
<tr>
<td>0x18</td>
<td>0x01</td>
<td>0xFF</td>
<td><strong>Generic Device Path Header</strong> – Type End of Hardware Device Path</td>
</tr>
<tr>
<td>0x19</td>
<td>0x01</td>
<td>0xFF</td>
<td>Sub type – End of Entire Device Path</td>
</tr>
<tr>
<td>0x1A</td>
<td>0x02</td>
<td>0x04</td>
<td>Length – 0x04 bytes</td>
</tr>
</tbody>
</table>

Another example is a USB Controller (interface number 0) that is connected to port 3 of a USB Hub Controller (interface number 0), and this USB Hub Controller is connected to the port 1 of the root hub. The shorthand notation for this device path is:

\[
\text{PciRoot(0)/PCI(31,2)/USB(1,0)/USB(3,0)}.
\]

**Table 10-20** shows the device path for this USB Controller.

**Table 10-21 Another USB Device Path Example**

<table>
<thead>
<tr>
<th>Byte Offset</th>
<th>Byte Length</th>
<th>Data</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x00</td>
<td>0x01</td>
<td>0x02</td>
<td><strong>Generic Device Path Header</strong> – Type ACPI Device Path</td>
</tr>
<tr>
<td>0x01</td>
<td>0x01</td>
<td>0x01</td>
<td>Sub type – ACPI Device Path</td>
</tr>
</tbody>
</table>
### 10.3.4.6 SATA Device Path

#### Table 10-22 SATA Device Path

<table>
<thead>
<tr>
<th>Mnemonic</th>
<th>Byte Offset</th>
<th>Byte Length</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Type</td>
<td>0</td>
<td>1</td>
<td>Type 3 – Messaging Device Path</td>
</tr>
<tr>
<td>Sub-Type</td>
<td>1</td>
<td>1</td>
<td>Sub-Type 18 – SATA</td>
</tr>
<tr>
<td>Length</td>
<td>2</td>
<td>2</td>
<td>Length of this structure in bytes. Length is 10 bytes.</td>
</tr>
<tr>
<td>HBA Port Number</td>
<td>4</td>
<td>2</td>
<td>The HBA port number that facilitates the connection to the device or a port multiplier. The value 0xFFFF is reserved.</td>
</tr>
<tr>
<td>Port Multiplier Port Number</td>
<td>6</td>
<td>2</td>
<td>The Port multiplier port number that facilitates the connection to the device. Must be set to 0xFFFF if the device is directly connected to the HBA.</td>
</tr>
<tr>
<td>Logical Unit Number</td>
<td>8</td>
<td>2</td>
<td>Logical Unit Number.</td>
</tr>
</tbody>
</table>
10.3.4.7 USB Device Paths (WWID)

This device path describes a USB device using its serial number.

Specifications, such as the USB Mass Storage class, bulk-only transport subclass, require that some portion of the suffix of the device’s serial number be unique with respect to the vendor and product id for the device. So, in order to avoid confusion and overlap of WWID’s, the interface’s class, subclass, and protocol are included.

Table 10-23 USB WWID Device Path

<table>
<thead>
<tr>
<th>Mnemonic</th>
<th>Byte Offset</th>
<th>Byte Length</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Type</td>
<td>0</td>
<td>1</td>
<td>Type 3 - Messaging Device Path</td>
</tr>
<tr>
<td>Sub-Type</td>
<td>1</td>
<td>1</td>
<td>Sub-Type 16 – USB WWID</td>
</tr>
<tr>
<td>Length</td>
<td>2</td>
<td>2</td>
<td>Length of this structure in bytes. Length is 10+</td>
</tr>
<tr>
<td>• Interface Number</td>
<td>4</td>
<td>2</td>
<td>USB interface number</td>
</tr>
<tr>
<td>• Device Vendor Id</td>
<td>6</td>
<td>2</td>
<td>USB vendor id of the device</td>
</tr>
<tr>
<td>• Device Product Id</td>
<td>8</td>
<td>2</td>
<td>USB product id of the device</td>
</tr>
<tr>
<td>• Serial Number</td>
<td>10</td>
<td>n</td>
<td>Last 64-or-fewer UTF-16 characters of the USB serial number. The length of the string is determined by the Length field less the offset of the Serial Number field (10)</td>
</tr>
</tbody>
</table>

Devices that do not have a serial number string must use with the USB Device Path (type 5) as described in Section 10.3.4.5.

Including the interface as part of this node allows distinction for multi-interface devices, e.g., an HID interface and a Mass Storage interface on the same device, or two Mass Storage interfaces. Section 3.1.2 defines special rules for processing the USB WWID Device Path. These special rules enable a device location to change and still have the system boot from the device.

10.3.4.8 Device Logical Unit

For some classes of devices, such as USB Mass Storage, it is necessary to specify the Logical Unit Number (LUN), since a single device may have multiple logical units. In order to boot from one of these logical units of the device, the Device Logical Unit device node is appended to the device path. The EFI path node subtype is defined, as in Table 10-24.

Table 10-24 Device Logical Unit

<table>
<thead>
<tr>
<th>Mnemonic</th>
<th>Byte Offset</th>
<th>Byte Length</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Type</td>
<td>0</td>
<td>1</td>
<td>Type 3 - Messaging Device Path</td>
</tr>
<tr>
<td>Sub-Type</td>
<td>1</td>
<td>1</td>
<td>Sub-Type 17 – Device Logical unit</td>
</tr>
<tr>
<td>Length</td>
<td>2</td>
<td>2</td>
<td>Length of this structure in bytes. Length is 5</td>
</tr>
<tr>
<td>LUN</td>
<td>4</td>
<td>1</td>
<td>Logical Unit Number for the interface</td>
</tr>
</tbody>
</table>
Section 3.1.2 defines special rules for processing the USB Class Device Path. These special rules enable a device location to change and still have the system recognize the device.

Section 3.3 defines how the \texttt{ConIn}, \texttt{ConOut}, and \texttt{ErrOut} variables are processed and contains special rules for processing the USB Class device path. These special rules allow all USB keyboards to be specified as valid input devices.

### 10.3.4.9 USB Device Path (Class)

Table 10-25 USB Class Device Path

<table>
<thead>
<tr>
<th>Mnemonic</th>
<th>Byte Offset</th>
<th>Byte Length</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Type</td>
<td>0</td>
<td>1</td>
<td>Type 3 - Messaging Device Path.</td>
</tr>
<tr>
<td>Sub-Type</td>
<td>1</td>
<td>1</td>
<td>Sub-Type 15 - USB Class.</td>
</tr>
<tr>
<td>Length</td>
<td>2</td>
<td>2</td>
<td>Length of this structure in bytes. Length is 11 bytes.</td>
</tr>
<tr>
<td>Vendor ID</td>
<td>4</td>
<td>2</td>
<td>Vendor ID assigned by USB-IF. A value of 0xFFFF will match any Vendor ID.</td>
</tr>
<tr>
<td>Product ID</td>
<td>6</td>
<td>2</td>
<td>Product ID assigned by USB-IF. A value of 0xFFFF will match any Product ID.</td>
</tr>
<tr>
<td>Device Class</td>
<td>8</td>
<td>1</td>
<td>The class code assigned by the USB-IF. A value of 0xFF will match any class code.</td>
</tr>
<tr>
<td>Device Subclass</td>
<td>9</td>
<td>1</td>
<td>The subclass code assigned by the USB-IF. A value of 0xFF will match any subclass code.</td>
</tr>
<tr>
<td>Device Protocol</td>
<td>10</td>
<td>1</td>
<td>The protocol code assigned by the USB-IF. A value of 0xFF will match any protocol code.</td>
</tr>
</tbody>
</table>

### 10.3.4.10 I\textsubscript{2}O Device Path

Table 10-26 I\textsubscript{2}O Device Path

<table>
<thead>
<tr>
<th>Mnemonic</th>
<th>Byte Offset</th>
<th>Byte Length</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Type</td>
<td>0</td>
<td>1</td>
<td>Type 3 – Messaging Device Path</td>
</tr>
<tr>
<td>Sub-Type</td>
<td>1</td>
<td>1</td>
<td>Sub-Type 6 – I2O Random Block Storage Class</td>
</tr>
<tr>
<td>Length</td>
<td>2</td>
<td>2</td>
<td>Length of this structure in bytes. Length is 8 bytes.</td>
</tr>
<tr>
<td>TID</td>
<td>4</td>
<td>4</td>
<td>Target ID (TID) for a device</td>
</tr>
</tbody>
</table>

### 10.3.4.11 MAC Address Device Path

Table 10-27 MAC Address Device Path

<table>
<thead>
<tr>
<th>Mnemonic</th>
<th>Byte Offset</th>
<th>Byte Length</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Type</td>
<td>0</td>
<td>1</td>
<td>Type 3 – Messaging Device Path</td>
</tr>
<tr>
<td>Sub-Type</td>
<td>1</td>
<td>1</td>
<td>Sub-Type 11 – MAC Address for a network interface</td>
</tr>
</tbody>
</table>
### 10.3.4.12 IPv4 Device Path

Previous versions of the specification only defined a 19 byte IPv4 device path. To access fields at off-set 19 or greater, the size of the device path must be checked first.

**Table 10-28 IPv4 Device Path**

<table>
<thead>
<tr>
<th>Mnemonic</th>
<th>Byte Offset</th>
<th>Byte Length</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Type</td>
<td>0</td>
<td>1</td>
<td>Type 3 – Messaging Device Path</td>
</tr>
<tr>
<td>Sub-Type</td>
<td>1</td>
<td>1</td>
<td>Sub-Type 12 – IPv4</td>
</tr>
<tr>
<td>Length</td>
<td>2</td>
<td>2</td>
<td>Length of this structure in bytes. Length is 27 bytes.</td>
</tr>
<tr>
<td>Local IP Address</td>
<td>4</td>
<td>4</td>
<td>The local IPv4 address</td>
</tr>
<tr>
<td>Remote IP Address</td>
<td>8</td>
<td>4</td>
<td>The remote IPv4 address</td>
</tr>
<tr>
<td>Local Port</td>
<td>12</td>
<td>2</td>
<td>The local port number</td>
</tr>
<tr>
<td>Remote Port</td>
<td>14</td>
<td>2</td>
<td>The remote port number</td>
</tr>
<tr>
<td>Protocol</td>
<td>16</td>
<td>2</td>
<td>The network protocol (i.e., UDP, TCP). See RFC 3232</td>
</tr>
<tr>
<td>StaticIPAddress</td>
<td>18</td>
<td>1</td>
<td>0x00 - The Source IP Address was assigned though DHCP 0x01 - The Source IP Address is statically bound</td>
</tr>
<tr>
<td>GatewayIPAddress</td>
<td>19</td>
<td>4</td>
<td>The Gateway IP Address</td>
</tr>
<tr>
<td>Subnet Mask</td>
<td>23</td>
<td>4</td>
<td>Subnet mask</td>
</tr>
</tbody>
</table>

### 10.3.4.13 IPv6 Device Path

**Table 10-29 IPv6 Device Path**

<table>
<thead>
<tr>
<th>Mnemonic</th>
<th>Byte Offset</th>
<th>Byte Length</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Type</td>
<td>0</td>
<td>1</td>
<td>Type 3 – Messaging Device Path</td>
</tr>
<tr>
<td>Sub-Type</td>
<td>1</td>
<td>1</td>
<td>Sub-Type 13 – IPv6</td>
</tr>
<tr>
<td>Length</td>
<td>2</td>
<td>2</td>
<td>Length of this structure in bytes. Length is 60 bytes.</td>
</tr>
<tr>
<td>Local IP Address</td>
<td>4</td>
<td>16</td>
<td>The local IPv6 address</td>
</tr>
<tr>
<td>Remote IP Address</td>
<td>20</td>
<td>16</td>
<td>The remote IPv6 address</td>
</tr>
<tr>
<td>Local Port</td>
<td>36</td>
<td>2</td>
<td>The local port number</td>
</tr>
<tr>
<td>Remote Port</td>
<td>38</td>
<td>2</td>
<td>The remote port number</td>
</tr>
<tr>
<td>Protocol</td>
<td>40</td>
<td>2</td>
<td>The network protocol (i.e., UDP, TCP). See RFC 3232</td>
</tr>
<tr>
<td>IPAddressOrigin</td>
<td>42</td>
<td>1</td>
<td>0x00 - The Local IP Address was manually configured. 0x01 - The Local IP Address is assigned through IPv6 stateless auto-configuration. 0x02 - The Local IP Address is assigned through IPv6 stateful configuration.</td>
</tr>
</tbody>
</table>
## 10.3.4.14 2. VLAN device path node

<table>
<thead>
<tr>
<th>Mnemonic</th>
<th>Byte Offset</th>
<th>Byte Length</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Type</td>
<td>0</td>
<td>1</td>
<td>Type 3 – Messaging Device Path</td>
</tr>
<tr>
<td>Sub-Type</td>
<td>1</td>
<td>1</td>
<td>Sub-Type 20 – Vlan (802.1q)</td>
</tr>
<tr>
<td>Length</td>
<td>2</td>
<td>2</td>
<td>Length of this device node</td>
</tr>
<tr>
<td>VlanId</td>
<td>4</td>
<td>2</td>
<td>VLAN identifier (0-4094)</td>
</tr>
</tbody>
</table>

## 10.3.4.15 InfiniBand Device Path

### Table 10-30 InfiniBand Device Path

<table>
<thead>
<tr>
<th>Mnemonic</th>
<th>Byte Offset</th>
<th>Byte Length</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Type</td>
<td>0</td>
<td>1</td>
<td>Type 3 – Messaging Device Path</td>
</tr>
<tr>
<td>Sub-Type</td>
<td>1</td>
<td>1</td>
<td>Sub-Type 9 – InfiniBand</td>
</tr>
<tr>
<td>Length</td>
<td>2</td>
<td>2</td>
<td>Length of this structure in bytes. Length is 48 bytes.</td>
</tr>
</tbody>
</table>
| Resource Flags          | 4           | 4           | Flags to help identify/manage InfiniBand device path elements:  
  • Bit 0 – IOC/Service (0b = IOC, 1b = Service)  
  • Bit 1 – Extend Boot Environment  
  • Bit 2 – Console Protocol  
  • Bit 3 – Storage Protocol  
  • Bit 4 – Network Protocol  
  All other bits are reserved. |
| PORT GID                | 8           | 16          | 128-bit Global Identifier for remote fabric port |
| IOC GUID/Service ID     | 24          | 8           | 64-bit unique identifier to remote IOC or server process.  
  Interpretation of field specified by Resource Flags (bit 0) |
| Target Port ID          | 32          | 8           | 64-bit persistent ID of remote IOC port |
| Device ID               | 40          | 8           | 64-bit persistent ID of remote device |

**Note:** The usage of the terms GUID and GID is per the InfiniBand Specification. The term GUID is not the same as the `EFI_GUID` type defined in this EFI Specification.

## 10.3.4.16 UART Device Path

### Table 10-31 UART Device Path

<table>
<thead>
<tr>
<th>Mnemonic</th>
<th>Byte Offset</th>
<th>Byte Length</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Type</td>
<td>0</td>
<td>1</td>
<td>Type 3 – Messaging Device Path</td>
</tr>
<tr>
<td>Sub-Type</td>
<td>1</td>
<td>1</td>
<td>Sub-Type 14 – UART</td>
</tr>
</tbody>
</table>
10.3.4.17 Vendor-Defined Messaging Device Path

Table 10-32 Vendor-Defined Messaging Device Path

<table>
<thead>
<tr>
<th>Mnemonic</th>
<th>Byte Offset</th>
<th>Byte Length</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Type</td>
<td>0</td>
<td>1</td>
<td>Type 3 – Messaging Device Path</td>
</tr>
<tr>
<td>Sub-Type</td>
<td>1</td>
<td>1</td>
<td>Sub-Type 10 – Vendor</td>
</tr>
<tr>
<td>Length</td>
<td>2</td>
<td>2</td>
<td>Length of this structure in bytes. Length is 20 + n bytes.</td>
</tr>
<tr>
<td>Vendor GUID</td>
<td>4</td>
<td>16</td>
<td>Vendor-assigned GUID that defines the data that follows</td>
</tr>
<tr>
<td>Vendor Defined Data</td>
<td>20</td>
<td>n</td>
<td>Vendor-defined variable size data</td>
</tr>
</tbody>
</table>

The following GUIDs are used with a Vendor-Defined Messaging Device Path to describe the transport protocol for use with PC-ANSI, VT-100, VT-100+, and VT-UTF8 terminals. Device paths can be constructed with this node as the last node in the device path. The rest of the device path describes the physical device that is being used to transmit and receive data. The PC-ANSI, VT-100, VT-100+, and VT-UTF8 GUIDs define the format of the data that is being sent through the physical device. Additional GUIDs can be generated to describe additional transport protocols.

```
#define EFI_PC_ANSI_GUID
{0xe0c14753,0xf9be,0x11d2,{0x9a,0x0c,0x00,0x27,0x3f,0xc1,0x4d }}
```

```
#define EFI_VT_100_GUID
{0xdfa66065,0xb419,0x11d3,{0x9a,0x2d,0x00,0x27,0x3f,0xc1,0x4d }}
```

```
#define EFI_VT_100_PLUS_GUID
{0x7baec70b,0x57e0,0x4c76,{0x8e,0x87,0x2f,0x9e,0x28,0x08,0x83,0x43 }}
```
#define EFI_VT_UTF8_GUID
{0xad15a0d6,0x8bec,0x4acf,{0xa0,0x73,0xd0,0x1d,0xe7,0x7e,0x2d,0x88}}

10.3.4.18 UART Flow Control Messaging Path

The UART messaging device path defined in the EFI 1.02 specification does not contain a provision for flow control. Therefore, a new device path node is needed to declare flow control characteristics. It is a vendor-defined messaging node which may be appended to the UART node in a device path. It has the following definition:

#define DEVICE_PATH_MESSAGING_UART_FLOW_CONTROL
{0x37499a9d,0x542f,0x4c89,{0xa0,0x26,0x35,0xda,0x14,0x20,0xe4}}

Table 10-33 UART Flow Control Messaging Device Path

<table>
<thead>
<tr>
<th>Mnemonic</th>
<th>Byte Offset</th>
<th>Byte Length</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Type</td>
<td>0</td>
<td>1</td>
<td>Type 3 — Messaging Device Path</td>
</tr>
<tr>
<td>Sub-Type</td>
<td>1</td>
<td>1</td>
<td>Sub-Type 10 — Vendor</td>
</tr>
<tr>
<td>Length</td>
<td>2</td>
<td>2</td>
<td>Length of this structure in bytes. Length is 24 bytes.</td>
</tr>
<tr>
<td>Vendor GUID</td>
<td>4</td>
<td>16</td>
<td><strong>DEVICE_PATH_MESSAGING_UART_FLOW_CONTROL</strong></td>
</tr>
<tr>
<td>Flow_Control_Map</td>
<td>20</td>
<td>4</td>
<td>Bitmap of supported flow control types.</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>• Bit 0 set indicates hardware flow control.</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>• Bit 1 set indicates Xon/Xoff flow control.</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>• All other bits are reserved and are clear.</td>
</tr>
</tbody>
</table>

A debugport driver that implements Xon/Xoff flow control would produce a device path similar to the following:

PciRoot(0)/Pci(0x1f,0)/ACPI(PNP0501,0)/UART(115200,N,8,1)/UartFlowCtrl(2)/DebugPort()

**Note:** If no bits are set in the Flow_Control_Map, this indicates there is no flow control and is equivalent to leaving the flow control node out of the device path completely.
10.3.4.19 Serial Attached SCSI (SAS) Device Path
This section defines the device node for Serial Attached SCSI (SAS) devices.

Table 10-34 Messaging Device Path Structure

<table>
<thead>
<tr>
<th>Mnemonic</th>
<th>Byte Offset</th>
<th>Byte Length</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Type</td>
<td>0</td>
<td>1</td>
<td>Type -3 Messaging</td>
</tr>
<tr>
<td>Sub Type</td>
<td>1</td>
<td>1</td>
<td>10 (Vendor)</td>
</tr>
<tr>
<td>Length</td>
<td>2</td>
<td>2</td>
<td>Length of this Structure.</td>
</tr>
<tr>
<td>Vendor GUID</td>
<td>4</td>
<td>16</td>
<td>d487d6db4-008b-11d9-afdc-001083ffca4d</td>
</tr>
<tr>
<td>Reserved</td>
<td>20</td>
<td>4</td>
<td>Reserved for future use.</td>
</tr>
<tr>
<td>SAS Address</td>
<td>24</td>
<td>8</td>
<td>SAS Address for Serial Attached SCSI Target.</td>
</tr>
<tr>
<td>Logical Unit Number</td>
<td>32</td>
<td>8</td>
<td>SAS Logical Unit Number.</td>
</tr>
<tr>
<td>SAS/SATA device and Topology Info</td>
<td>40</td>
<td>2</td>
<td>More Information about the device and its interconnect</td>
</tr>
<tr>
<td>Relative Target Port</td>
<td>42</td>
<td>2</td>
<td>Relative Target Port (RTP)</td>
</tr>
</tbody>
</table>

Summary
The device node represented by the structure in Table 10-34 (above) shall be appended after the Hardware Device Path node in the device path.

There are two cases for boot devices connected with SAS HBA’s. Each of the cases is described below with an example of the expected Device Path for these.

  
PciRoot(0)/PCI(1,0)/Sas(0x31000004CF13F6BD, 0)
  
  The first 64-bit number represents the SAS address of the target SAS device.
  
  The second number is the boot LUN of the target SAS device.
  
  The third number is the Relative Target Port (RTP)

- SATA Device connected directly to a HBA port.
  
PciRoot(0)/PCI(1,0)/Sas(0x31000004CF13F6BD)
  
  The first number represents either a real SAS address reserved by the HBA for above connections, or a fake but unique SAS address generated by the HBA to represent the SATA device.

10.3.4.19.1 Device and Topology Information
First Byte (At offset 40 into the structure):

Bits 0:3:

Value 0x0 -> No Additional Information about device topology.

Value 0x1 -> More Information about device topology valid in this byte.

Value 0x2 -> More Information about device topology valid in this and next 1 byte.

Values 0x3 thru 0xF -> Reserved.
Bits 4:5: Device Type (Valid only if the More Information field above is non-zero)
Value 0x0 -> SAS Internal Device
Value 0x1 -> SATA Internal Device
Value 0x2 -> SAS External Device
Value 0x3 -> SATA External Device

Bits 6:7: Topology / Interconnect (Valid only if the More Information field above is non-zero)
Value 0x0 -> Direct Connect (Connected directly with the HBA Port/Phy)
Value 0x1 -> Expander Connect (Connected thru/via one or more Expanders)
Value 0x2 and 0x3 > Reserved

10.3.4.19.2 Device and Topology Information
Second Byte (At offset 41 into the structure). Valid only if bits 0-3 of More Information in Byte 40 have a value of 2:
Bits 0-7: Internal Drive/Bay Id (Only applicable if Internal Drive is indicated in Device Type)
Value 0x0 thru 0xFF -> Drive 1 thru Drive 256

10.3.4.19.3 Relative Target Port
At offset 42 into the structure:
This two-byte field shall contain the “Relative Target Port” of the target SAS port. Relative Target Port can be obtained by performing an INQUIRY command to VPD page 0x83 in the target. Implementation of RTP is mandatory for SAS targets as defined in Section 10.2.10 of sas1r07 specification (or later).

Note: If a LUN is seen thru multiple RTPs in a given target, then the UEFI driver shall create separate device path instances for both paths. RTP in the device path shall distinguish these two device path instantiations.

Note: Changing the values of the SAS/SATA device topology information or the RTP fields of the device path will make UEFI think this is a different device.

10.3.4.19.4 Examples Of Correct Device Path Display Format
Case 1: When Additional Information is not Valid or Not Present (Bits 0:3 of Byte 40 have a value of 0)
\[ \text{PciRoot}(0)/\text{PCI}(1,0)/\text{SAS}(0x310000004CF13F6BD, 0) \]

Case 2: When Additional Information is Valid and present (Bits 0:3 of Byte 40 have a value of 1 or 2)
- If Bits 4-5 of Byte 40 (Device and Topology information) indicate an SAS device (Internal or External) i.e., has values 0x0 or 0x2, then the following format shall be used.
  \[ \text{PciRoot}(0)/\text{PCI}(1,0)/\text{SAS}(0x310000004CF13F6BD, 0, \text{SAS}) \]
- If Bits 4-5 of Byte 40 (Device and Topology information) indicate a SATA device (Internal or External) i.e., has a value of 0x1 or 0x3, then the following format shall be used.
10.3.4.20 Serial Attached SCSI (SAS) Ex Device Path

This section defines the extended device node for Serial Attached SCSI (SAS) devices. In this device path the SAS Address and LUN are now defined as arrays to remove the need to endian swap the values.

### Table 10-35 Messaging Device Path Structure

<table>
<thead>
<tr>
<th>Mnemonic</th>
<th>Byte Offset</th>
<th>Byte Length</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Type</td>
<td>0</td>
<td>1</td>
<td>Type -3 Messaging</td>
</tr>
<tr>
<td>Sub Type</td>
<td>1</td>
<td>1</td>
<td>Sub-type 22 SAS Ex</td>
</tr>
<tr>
<td>Length</td>
<td>2</td>
<td>2</td>
<td>Length of this Structure. 32 bytes</td>
</tr>
<tr>
<td>SAS Address</td>
<td>4</td>
<td>8</td>
<td>8-byte array of the SAS Address for Serial Attached SCSI Target Port.</td>
</tr>
<tr>
<td>Logical Unit Number</td>
<td>20</td>
<td>8</td>
<td>8-byte array of the SAS Logical Unit Number.</td>
</tr>
<tr>
<td>SAS/SATA device and Topology</td>
<td>28</td>
<td>2</td>
<td>More Information about the device and its interconnect</td>
</tr>
<tr>
<td>Info</td>
<td>30</td>
<td>2</td>
<td>Relative Target Port (RTP)</td>
</tr>
</tbody>
</table>

The SAS Ex device path clarifies the definition of the Logical Unit Number field to conform with the T-10 SCSI Architecture Model 4 specification. The 8 byte Logical Unit Number field in the device path must conform with a logical unit number returned by a SCSI REPORT LUNS command.

When the SAS Device Path Ex is used with the Extended SCSI Pass Thru Protocol, the UINT64 LUN must be converted to the eight byte array Logical Unit Number field in the device path by treating the eight byte array as an EFI UINT64. For example, a Logical Unit Number array of { 0,1,2,3,4,5,6,7 } becomes a UINT64 of 0x0706050403020100.

When an application client displays or otherwise makes a 64-bit LUN (8 byte array) visible to a user, it should be done in conformance with SAM-4. SAM-4 requires a LUN to be displayed in hexadecimal format with byte 0 first (i.e., on the left) and byte 7 last (i.e., on the right) regardless of the internal representation of the LUN. UEFI defines all data structures a “little endian” and SCSI defines all data structures as “big endian”.

10.3.4.21 iSCSI Device Path

### Table 10-36 iSCSI Device Path Node (Base Information)

<table>
<thead>
<tr>
<th>Mnemonic</th>
<th>Byte Offset</th>
<th>Byte Length</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Type</td>
<td>0</td>
<td>1</td>
<td>Type 3 – Messaging Device Path</td>
</tr>
<tr>
<td>Sub-Type</td>
<td>1</td>
<td>1</td>
<td>Sub-Type 19 – (iSCSI)</td>
</tr>
<tr>
<td>Length</td>
<td>2</td>
<td>2</td>
<td>Length of this structure in bytes. Length is (18 + n) bytes</td>
</tr>
<tr>
<td>Protocol</td>
<td>4</td>
<td>2</td>
<td>Network Protocol (0 = TCP, 1+ = reserved)</td>
</tr>
<tr>
<td>Options</td>
<td>6</td>
<td>2</td>
<td>iSCSI Login Options</td>
</tr>
</tbody>
</table>
10.3.4.21.1 iSCSI Login Options

The iSCSI Device Node Options describe the iSCSI login options for the key values:

- **Bits 0-1:**
  - 0 = No Header Digest
  - 2 = Header Digest Using CRC32C

- **Bits 2-3:**
  - 0 = No Data Digest
  - 2 = Data Digest Using CRC32C

- **Bits 4-9:**
  - Reserved for future use

- **Bits 10-11:**
  - 0 = AuthMethod_CHAP
  - 2 = AuthMethod_None

- **Bit 12:**
  - 0 = CHAP_BI
  - 1 = CHAP_UNI

For each specific login key, none, some or all of the defined values may be configured. If none of the options are defined for a specific key, the iSCSI driver shall propose “None” as the value. If more than one option is configured for a specific key, all the configured values will be proposed (ordering of the values is implementation dependent).

- **Portal Group Tag:** defines the iSCSI portal group the initiator intends to establish a session with.

- **Logical Unit Number:** defines the 8 byte SCSI LUN. The Logical Unit Number field must conform to the T-10 SCSI Architecture Model 4 specification. The 8 byte Logical Unit Number field in the device path must conform with a logical unit number returned by a SCSI REPORT LUNS command.

- **iSCSI Target Name:** defines the iSCSI Target Name for the iSCSI Node. The size of the iSCSI Target Name can be up to a maximum of 223 bytes.

10.3.4.21.2 Device Path Examples

Some examples for the Device Path for the case the boot device connected to iSCSI bootable controller:
With IPv4 configuration:

```
PciRoot(0)/Pci(19|0)/Mac(001320F5FA77,0x01)/
IPv4(192.168.0.100,TCP,Static,192.168.0.1)/
    iSCSI(iqn.1991-05.com.microsoft:iscsitarget-iscsidisk-target,0x1,0x0,None,None,None,TCP)/
    HD(1,GPT,15E39A00-1DD2-1000-BD7F-00A0C92408FC,0x22,0x2710000)
```

Table 10-37 IPv4 configuration

<table>
<thead>
<tr>
<th>Byte Offset</th>
<th>Byte Length</th>
<th>Data</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x00</td>
<td>1</td>
<td>0x02</td>
<td><strong>Generic Device Path Header</strong> – Type ACPI Device Path</td>
</tr>
<tr>
<td>0x01</td>
<td>1</td>
<td>0x01</td>
<td>Sub type – ACPI Device Path</td>
</tr>
<tr>
<td>0x02</td>
<td>2</td>
<td>0x0C</td>
<td>Length – 0x0C bytes</td>
</tr>
<tr>
<td>0x04</td>
<td>4</td>
<td>0x41D0, 0x0A03</td>
<td>_HID PNP0A03 – 0x41D0 represents the compressed string 'PNP' and is encoded in the low order bytes. The compression method is described in the ACPI Specification.</td>
</tr>
<tr>
<td>0x08</td>
<td>4</td>
<td>0x0000</td>
<td>_UID</td>
</tr>
<tr>
<td>0x0C</td>
<td>1</td>
<td>0x01</td>
<td><strong>Generic Device Path Header</strong> – Type Hardware Device Path</td>
</tr>
<tr>
<td>0x0D</td>
<td>1</td>
<td>0x01</td>
<td>Sub type – PCI</td>
</tr>
<tr>
<td>0x0E</td>
<td>2</td>
<td>0x06</td>
<td>Length – 0x06 bytes</td>
</tr>
<tr>
<td>0x10</td>
<td>1</td>
<td>0x0</td>
<td>PCI Function</td>
</tr>
<tr>
<td>0x11</td>
<td>1</td>
<td>0x19</td>
<td>PCI Device</td>
</tr>
<tr>
<td>0x12</td>
<td>1</td>
<td>0x03</td>
<td><strong>Generic Device Path Header</strong> – Messaging Device Path</td>
</tr>
<tr>
<td>0x13</td>
<td>1</td>
<td>0x08</td>
<td>Sub type – MAC Address Device path</td>
</tr>
<tr>
<td>0x14</td>
<td>2</td>
<td>0x25</td>
<td>Length – 0x25</td>
</tr>
<tr>
<td>Address</td>
<td>Size</td>
<td>Data</td>
<td>Description</td>
</tr>
<tr>
<td>---------</td>
<td>------</td>
<td>------</td>
<td>-------------</td>
</tr>
<tr>
<td>0x16</td>
<td>32</td>
<td>0x00, 0x13, 0x20, 0xF5, 0xFA, 0x77, 0x00,...</td>
<td>MAC address for a network interface padded with zeros</td>
</tr>
<tr>
<td>0x36</td>
<td>1</td>
<td>0x01</td>
<td>Network Interface Type - other</td>
</tr>
<tr>
<td>0x37</td>
<td>1</td>
<td>0x03</td>
<td>Generic Device Path Header – Messaging Device Path</td>
</tr>
<tr>
<td>0x38</td>
<td>1</td>
<td>0x0c</td>
<td>Sub type – IPv4</td>
</tr>
<tr>
<td>0x39</td>
<td>2</td>
<td>0x1B</td>
<td>Length – 27</td>
</tr>
<tr>
<td>0x3b</td>
<td>4</td>
<td>0xC0, 0xA8, 0x00, 0x01</td>
<td>Local IPv4 address – 192.168.0.1</td>
</tr>
<tr>
<td>0x3F</td>
<td>4</td>
<td>0xC0, 0xA8, 0x00, 0x64</td>
<td>Remote IPv4 address – 192.168.0.100</td>
</tr>
<tr>
<td>0x43</td>
<td>2</td>
<td>0x0000</td>
<td>Local Port Number – 0</td>
</tr>
<tr>
<td>0x45</td>
<td>2</td>
<td>0x0CBC</td>
<td>Remote Port Number – 3260</td>
</tr>
<tr>
<td>0x47</td>
<td>2</td>
<td>0x6</td>
<td>Network Protocol. See RFC 3232. TCP</td>
</tr>
<tr>
<td>0x49</td>
<td>1</td>
<td>1</td>
<td>Static IP Address</td>
</tr>
<tr>
<td>0x4A</td>
<td>4</td>
<td></td>
<td>Gateway IP Address</td>
</tr>
<tr>
<td>0x4E</td>
<td>4</td>
<td></td>
<td>Subnet mask</td>
</tr>
<tr>
<td>0x52</td>
<td>1</td>
<td>0x03</td>
<td>Generic Device Path Header – Messaging Device Path</td>
</tr>
<tr>
<td>0x53</td>
<td>1</td>
<td>0x13</td>
<td>Sub type – iSCSI</td>
</tr>
<tr>
<td>0x54</td>
<td>2</td>
<td>0x49</td>
<td>Length – 0x49</td>
</tr>
<tr>
<td>0x56</td>
<td>2</td>
<td>0x00</td>
<td>Network Protocol</td>
</tr>
<tr>
<td>0x58</td>
<td>2</td>
<td>0x800</td>
<td>iSCSI Login Options – AuthMethod_None</td>
</tr>
<tr>
<td>Offset (Hex)</td>
<td>Length</td>
<td>Value</td>
<td>Description</td>
</tr>
<tr>
<td>-------------</td>
<td>--------</td>
<td>-------</td>
<td>-------------</td>
</tr>
<tr>
<td>0x5A</td>
<td>8</td>
<td>0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00</td>
<td>iSCSI LUN</td>
</tr>
<tr>
<td>0x62</td>
<td>2</td>
<td>0x01</td>
<td>Target Portal group tag</td>
</tr>
<tr>
<td>0x64</td>
<td>55</td>
<td>0x69, 0x71, 0x6E, 0x2E, 0x31, 0x39, 0x39, 0x31, 0x2D, 0x30, 0x35, 0x2E, 0x63, 0x6F, 0x6D, 0x2E, 0x6D, 0x69, 0x63, 0x72, 0x6F, 0x73, 0x6F, 0x66, 0x74,</td>
<td>iSCSI node name.</td>
</tr>
<tr>
<td>0x64 (cont.)</td>
<td>55</td>
<td>0x3A, 0x69, 0x73, 0x63, 0x73, 0x69, 0x74, 0x61, 0x72, 0x67, 0x65, 0x74, 0x2D, 0x69, 0x73, 0x63, 0x73, 0x69, 0x64, 0x69, 0x73, 0x6B, 0x2D, 0x74, 0x61, 0x72, 0x67, 0x65, 0x74, 0x00</td>
<td>iSCSI node name (cont.)</td>
</tr>
<tr>
<td>0x9B</td>
<td>1</td>
<td>0x04</td>
<td>Generic Device Path Header – Media Device Path</td>
</tr>
<tr>
<td>0x9C</td>
<td>1</td>
<td>0x01</td>
<td>Sub type – Hard Drive</td>
</tr>
<tr>
<td>0x9D</td>
<td>2</td>
<td>0x2A</td>
<td>Length – 0x2a</td>
</tr>
<tr>
<td>0x9F</td>
<td>4</td>
<td>0x1</td>
<td>Partition Number</td>
</tr>
<tr>
<td>0xA3</td>
<td>8</td>
<td>0x22</td>
<td>Partition Start</td>
</tr>
<tr>
<td>0xAB</td>
<td>8</td>
<td>0x2710000</td>
<td>Partition Size</td>
</tr>
<tr>
<td>0xB3</td>
<td>16</td>
<td>0x00, 0x9A, 0xE3, 0x15, 0xD2, 0x1D, 0x00, 0x10, 0x8D, 0x7F, 0x00, 0xA0, 0xC9, 0x24, 0x08, 0xFC</td>
<td>Partition Signature</td>
</tr>
<tr>
<td>0xC3</td>
<td>1</td>
<td>0x02</td>
<td>Partition Format – GPT</td>
</tr>
<tr>
<td>0xC4</td>
<td>1</td>
<td>0x02</td>
<td>Signature Type – GUID</td>
</tr>
<tr>
<td>0xC5</td>
<td>1</td>
<td>0xFF</td>
<td>Generic Device Path Header – Type End of Hardware Device Path</td>
</tr>
<tr>
<td>0xC6</td>
<td>1</td>
<td>0xFF</td>
<td>Sub type – End of Entire Device Path</td>
</tr>
<tr>
<td>0xC7</td>
<td>2</td>
<td>0x04</td>
<td>Length – 0x04 bytes</td>
</tr>
</tbody>
</table>
• With IPv6 configuration:
  \( \text{PciRoot}(0x0)/\text{Pci}(0x1C,0x2)/\text{Pci}(0x0,0x0)/\text{MAC}(001517215593,0x0)/\text{IPv6}(2001:4898:000A:1005:95A6:EE6C:BED3:4859,\text{TCPDHCP},2001:4898:000A:1005:0215:17FF:FE21:5593)/\text{iSCSI}(\text{iqn.1991-05.com.microsoft:iscsiipv6-ipv6test-target},0x1,0x0,\text{None},\text{None},\text{None},\text{TCP})/\text{HD}(1,\text{MBR},0xA0021243,0x800,0x2EE000) \)

<table>
<thead>
<tr>
<th>Table 10-38 IPv6 configuration</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Byte Offset</strong></td>
</tr>
<tr>
<td>0x00</td>
</tr>
<tr>
<td>0x01</td>
</tr>
<tr>
<td>0x02</td>
</tr>
<tr>
<td>0x04</td>
</tr>
<tr>
<td>0x08</td>
</tr>
<tr>
<td>0x0C</td>
</tr>
<tr>
<td>0x0D</td>
</tr>
<tr>
<td>0x0E</td>
</tr>
<tr>
<td>0x10</td>
</tr>
<tr>
<td>0x11</td>
</tr>
<tr>
<td>0x12</td>
</tr>
<tr>
<td>0x13</td>
</tr>
<tr>
<td>0x14</td>
</tr>
<tr>
<td>0x16</td>
</tr>
<tr>
<td>0x17</td>
</tr>
<tr>
<td>0x18</td>
</tr>
<tr>
<td>0x19</td>
</tr>
<tr>
<td>0x1A</td>
</tr>
<tr>
<td>Byte Offset</td>
</tr>
<tr>
<td>-------------</td>
</tr>
<tr>
<td>0x1C</td>
</tr>
<tr>
<td>0x3C</td>
</tr>
<tr>
<td>0x3D</td>
</tr>
<tr>
<td>0x3E</td>
</tr>
<tr>
<td>0x3F</td>
</tr>
<tr>
<td>0x41</td>
</tr>
<tr>
<td>0x51</td>
</tr>
<tr>
<td>0x61</td>
</tr>
<tr>
<td>Byte Offset</td>
</tr>
<tr>
<td>-------------</td>
</tr>
<tr>
<td>0x63</td>
</tr>
<tr>
<td>0x65</td>
</tr>
<tr>
<td>0x66</td>
</tr>
<tr>
<td>0x67</td>
</tr>
<tr>
<td>0x68</td>
</tr>
<tr>
<td>0x78</td>
</tr>
<tr>
<td>0x79</td>
</tr>
<tr>
<td>0x7A</td>
</tr>
<tr>
<td>0x7C</td>
</tr>
<tr>
<td>0x7E</td>
</tr>
<tr>
<td>0x81</td>
</tr>
<tr>
<td>0x89</td>
</tr>
<tr>
<td>0x8B</td>
</tr>
<tr>
<td>(cont.)</td>
</tr>
<tr>
<td>0x8B (cont.)</td>
</tr>
<tr>
<td>0xBF</td>
</tr>
<tr>
<td>0xC0</td>
</tr>
</tbody>
</table>
10.3.4.22 NVM Express namespace messaging device path node

Table 10-39 NVM Express Namespace Device Path

<table>
<thead>
<tr>
<th>Mnemonic</th>
<th>Byte Offset</th>
<th>Byte Length</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Type</td>
<td>0</td>
<td>1</td>
<td>Type 3 – Messaging Device Path</td>
</tr>
<tr>
<td>Sub-Type</td>
<td>1</td>
<td>1</td>
<td>Sub Type 23 – NVM Express Namespace</td>
</tr>
<tr>
<td>Length</td>
<td>2</td>
<td>2</td>
<td>Length of this structure in bytes. Length is 16 bytes.</td>
</tr>
<tr>
<td>Namespace Identifier</td>
<td>4</td>
<td>4</td>
<td>Namespace identifier (NSID). The values of 0 and 0xFFFFFFFF are invalid.</td>
</tr>
<tr>
<td>IEEE Extended Unique Identifier</td>
<td>8</td>
<td>8</td>
<td>This field contains the IEEE Extended Unique Identifier (EUI-64). Devices without an EUI-64 value must initialize this field with a value of 0.</td>
</tr>
</tbody>
</table>

Refer to the latest NVM Express specification for descriptions of Namespace Identifier (NSID) and IEEE Extended Unique Identifier (EUI-64): See “Links to UEFI-Related Documents” (http://uefi.org/uefi) under the headings “NVM Express Specification”.

**Table 10-39 NVM Express Namespace Device Path**

<table>
<thead>
<tr>
<th>Byte Offset</th>
<th>Byte Length</th>
<th>Data</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>0xC1</td>
<td>2</td>
<td>0x2A</td>
<td>Length – 0x2a</td>
</tr>
<tr>
<td>0xC3</td>
<td>4</td>
<td>0x1</td>
<td>Partition Number</td>
</tr>
<tr>
<td>0xC7</td>
<td>8</td>
<td>0x800</td>
<td>Partition Start</td>
</tr>
<tr>
<td>0xCF</td>
<td>8</td>
<td>0x2EE000</td>
<td>Partition Size</td>
</tr>
<tr>
<td>0xDF</td>
<td>16</td>
<td>0x43, 0x12, 0x02, 0xA0, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00</td>
<td>Partition Signature</td>
</tr>
<tr>
<td>0xEF</td>
<td>1</td>
<td>0x01</td>
<td>Partition Format – MBR</td>
</tr>
<tr>
<td>0xF0</td>
<td>1</td>
<td>0x01</td>
<td>Signature Type – 32bit signature</td>
</tr>
<tr>
<td>0xF1</td>
<td>1</td>
<td>0xFF</td>
<td>Generic Device Path Header – Type End of Hardware Device Path</td>
</tr>
<tr>
<td>0xF2</td>
<td>1</td>
<td>0xFF</td>
<td>Sub type – End of Entire Device Path</td>
</tr>
<tr>
<td>0xF3</td>
<td>2</td>
<td>0x04</td>
<td>Length – 0x04 bytes</td>
</tr>
</tbody>
</table>
10.3.4.23 Uniform Resource Identifiers (URI) Device Path

Refer to RFC 3986 for details on the URI contents.

Table 10-40 URI Device Path

<table>
<thead>
<tr>
<th>Mnemonic</th>
<th>Byte Offset</th>
<th>Byte Length</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Type</td>
<td>0</td>
<td>1</td>
<td>Type 3 – Messaging Device Path</td>
</tr>
<tr>
<td>Sub-Type</td>
<td>1</td>
<td>1</td>
<td>Sub Type 24 – Universal Resource Identifier (URI) Device Path</td>
</tr>
<tr>
<td>Length</td>
<td>2</td>
<td>2</td>
<td>Length of this structure in bytes. Length is 4 + n bytes.</td>
</tr>
<tr>
<td>...</td>
<td>4</td>
<td>n</td>
<td>Instance of the URI pursuant to RFC 3986. For an empty URI, Length is 4 bytes.</td>
</tr>
</tbody>
</table>

10.3.4.24 UFS (Universal Flash Storage) device messaging device path node

Table 10-41 UFS Device Path

<table>
<thead>
<tr>
<th>Mnemonic</th>
<th>Byte Offset</th>
<th>Byte Length</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Type</td>
<td>0</td>
<td>1</td>
<td>Type 3 – Messaging Device Path</td>
</tr>
<tr>
<td>Sub-Type</td>
<td>1</td>
<td>1</td>
<td>Sub-Type 25 – UFS</td>
</tr>
<tr>
<td>Length</td>
<td>2</td>
<td>2</td>
<td>Length of this structure in bytes. Length is 6 bytes.</td>
</tr>
<tr>
<td>Target ID</td>
<td>4</td>
<td>1</td>
<td>Target ID on the UFS interface (PUN).</td>
</tr>
<tr>
<td>LUN</td>
<td>5</td>
<td>1</td>
<td>Logical Unit Number (LUN).</td>
</tr>
</tbody>
</table>

Refer to the UFS 2.0 specification for additional LUN descriptions: See “Links to UEFI-Related Documents” (http://uefi.org/uefi) under the heading “UFS 2.0 Specification”.

- **PUN field**: According to current available UFS 2.0 spec, the topology is one device per UFS port. A topology to support multiple devices on a single interface is planned for future revision. So suggest to reserve/introduce this field to support multiple devices per UFS port. This value should be 0 for current UFS2.0 spec compliance.

- **LUN field**: This field is used to specify up to 8 normal LUNs(0-7) and 4 well-known LUNs(81h, D0h, B0h and C4h). For those well-known LUNs, the BIT7 is set. See Figure 10.2 of UFS 2.0 spec for details.

10.3.4.25 SD (Secure Digital) Device Path

Table 10-42 SD Device Path

<table>
<thead>
<tr>
<th>Mnemonic</th>
<th>Byte Offset</th>
<th>Byte Length</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Type</td>
<td>0</td>
<td>1</td>
<td>Type 3 – Messaging Device Path</td>
</tr>
<tr>
<td>Sub-Type</td>
<td>1</td>
<td>1</td>
<td>Sub-Type 26 – SD</td>
</tr>
<tr>
<td>Length</td>
<td>2</td>
<td>2</td>
<td>Length of this structure in bytes. Length is 5 bytes.</td>
</tr>
<tr>
<td>Slot Number</td>
<td>4</td>
<td>1</td>
<td>Slot Number</td>
</tr>
</tbody>
</table>
### 10.3.4.26 EFI Bluetooth Device Path

Table 10-43 Bluetooth Device Path

<table>
<thead>
<tr>
<th>Mnemonic</th>
<th>Byte Offset</th>
<th>Byte Length</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Type</td>
<td>0</td>
<td>1</td>
<td>Type 3 – Messaging Device Path</td>
</tr>
<tr>
<td>Sub-Type</td>
<td>1</td>
<td>1</td>
<td>Sub-Type 27 – Bluetooth</td>
</tr>
<tr>
<td>Length</td>
<td>2</td>
<td>2</td>
<td>Length of this structure in bytes. Length is 10 bytes.</td>
</tr>
<tr>
<td>Bluetooth Device Address</td>
<td>4</td>
<td>6</td>
<td>48-bit Bluetooth device address</td>
</tr>
</tbody>
</table>

### 10.3.4.27 Wireless Device Path

Table 10-44 Wi-Fi Device Path

<table>
<thead>
<tr>
<th>Mnemonic</th>
<th>Byte Offset</th>
<th>Byte Length</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Type</td>
<td>0</td>
<td>1</td>
<td>Type 3 – Messaging Device Path</td>
</tr>
<tr>
<td>Sub-Type</td>
<td>1</td>
<td>1</td>
<td>Sub Type 28 – Wi-Fi Device Path</td>
</tr>
<tr>
<td>Length</td>
<td>2</td>
<td>2</td>
<td>Length of this structure in bytes. Length is 36 bytes.</td>
</tr>
<tr>
<td>SSID</td>
<td>4</td>
<td>32</td>
<td>SSID in octet string</td>
</tr>
</tbody>
</table>

### 10.3.4.28 eMMC (Embedded Multi-Media Card) Device Path

Table 10-45 eMMC Device Path

<table>
<thead>
<tr>
<th>Mnemonic</th>
<th>Byte Offset</th>
<th>Byte Length</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Type</td>
<td>0</td>
<td>1</td>
<td>Type 3 – Messaging Device Path</td>
</tr>
<tr>
<td>Sub-Type</td>
<td>1</td>
<td>1</td>
<td>Sub-Type 29 – eMMC</td>
</tr>
<tr>
<td>Length</td>
<td>2</td>
<td>2</td>
<td>Length of this structure in bytes. Length is 5 bytes.</td>
</tr>
<tr>
<td>Slot Number</td>
<td>4</td>
<td>1</td>
<td>Slot Number</td>
</tr>
</tbody>
</table>

### 10.3.4.29 EFI BluetoothLE Device Path

Table 10-46 EFI BluetoothLE Device Path

<table>
<thead>
<tr>
<th>Mnemonic</th>
<th>Byte Offset</th>
<th>Byte Length</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Type</td>
<td>0</td>
<td>1</td>
<td>Type 3 – Messaging Device Path</td>
</tr>
<tr>
<td>Sub-Type</td>
<td>1</td>
<td>1</td>
<td>Sub-Type 30 – BluetoothLE</td>
</tr>
<tr>
<td>Length</td>
<td>2</td>
<td>2</td>
<td>Length of this structure in bytes. Length is 11 bytes.</td>
</tr>
<tr>
<td>Bluetooth Device Address</td>
<td>4</td>
<td>6</td>
<td>48-bit Bluetooth device address</td>
</tr>
<tr>
<td>Address Type</td>
<td>10</td>
<td>1</td>
<td>0x00 – Public Device Address</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>0x01 – Random Device Address</td>
</tr>
</tbody>
</table>
10.3.4.30 DNS Device Path

Table 10-47 DNS Device Path

<table>
<thead>
<tr>
<th>Mnemonic</th>
<th>Byte Offset</th>
<th>Byte Length</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Type</td>
<td>0</td>
<td>1</td>
<td>Type 3 – Messaging Device Path</td>
</tr>
<tr>
<td>Sub-Type</td>
<td>1</td>
<td>1</td>
<td>Sub-Type 31 – DNS Device Path</td>
</tr>
<tr>
<td>Length</td>
<td>2</td>
<td>2</td>
<td>Length of this structure in bytes. Length is 5 + n bytes.</td>
</tr>
<tr>
<td>IsIPv6</td>
<td>4</td>
<td>1</td>
<td>0x00 – The DNS server address is IPv4 address. 0x01 – The DNS server address is IPv6 address.</td>
</tr>
<tr>
<td>...</td>
<td>5</td>
<td>n</td>
<td>One or more instances of the DNS server address in EFI_IP_ADDRESS.</td>
</tr>
</tbody>
</table>

10.3.4.31 NVDIMM Namespace path

This device path describes a bootable NVDIMM namespace that is defined by a namespace label. The presence of this type of device path indicates that UEFI supports booting to the namespace including any address abstraction specified by the namespace label. Refer to the NVDIMM Label Protocol section to retrieve relevant details about the namespace.

Table 10-48 NVDIMM Namespace Device Path

<table>
<thead>
<tr>
<th>Mnemonic</th>
<th>Byte Offset</th>
<th>Byte Length</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Type</td>
<td>0</td>
<td>1</td>
<td>Type 3 - Messaging Device</td>
</tr>
<tr>
<td>Sub-Type</td>
<td>1</td>
<td>1</td>
<td>Sub-type 32 - NVDIMM Namespace</td>
</tr>
<tr>
<td>Length</td>
<td>2</td>
<td>2</td>
<td>20 - Single namespace UUID is supported.</td>
</tr>
<tr>
<td>Uuid</td>
<td>4</td>
<td>16</td>
<td>Namespace unique label identifier UUID. See the Uuid description in the NVDIMM Label Protocol - Label definitions section for details on this field.</td>
</tr>
</tbody>
</table>

10.3.4.32 REST Service Device Path

<table>
<thead>
<tr>
<th>Mnemonic</th>
<th>Byte Offset</th>
<th>Byte Length</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Type</td>
<td>0</td>
<td>1</td>
<td>Type 0x03 – Messaging Device Path</td>
</tr>
<tr>
<td>Sub-Type</td>
<td>1</td>
<td>1</td>
<td>Sub-Type 32– REST Service Device Path</td>
</tr>
<tr>
<td>Length</td>
<td>2</td>
<td>2</td>
<td>Length of this structure in bytes. Length is 6 bytes.</td>
</tr>
<tr>
<td>REST Service</td>
<td>4</td>
<td>1</td>
<td>0x01 = Redfish REST Service 0x02 = OData REST Service</td>
</tr>
<tr>
<td>Access Mode</td>
<td>5</td>
<td>1</td>
<td>(0x01) In-Band REST Service, (0x02) Out-of-band REST Service.</td>
</tr>
</tbody>
</table>

Device path example of Out-of-band Redfish REST Service through NIC:

PciRoot(0x2)/Pci(0x2,0x0)/Pci(0x0,0x0)/MAC(FD19FA100672,0x0)/IPv4(0.0.0.0,0x0,DHCP,0.0.0.0,0.0.0.0,0.0.0.0)/RestService(1,2)
Below is vendor-specific REST Service Device Path

<table>
<thead>
<tr>
<th>Mnemonic</th>
<th>Byte Offset</th>
<th>Byte Length</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Type</td>
<td>0</td>
<td>1</td>
<td>Type 0x03 – Messaging Device Path</td>
</tr>
<tr>
<td>Sub-Type</td>
<td>1</td>
<td>1</td>
<td>Sub-Type 32– REST Service Device Path</td>
</tr>
<tr>
<td>Length</td>
<td>2</td>
<td>2</td>
<td>Length of this structure in bytes. Length is 21 + n bytes.</td>
</tr>
<tr>
<td>REST Service</td>
<td>4</td>
<td>1</td>
<td>0xFF = Vendor specific REST Service</td>
</tr>
<tr>
<td>Access Mode</td>
<td>5</td>
<td>1</td>
<td>(0x01) In-Band REST Service, (0x02) Out-of-band REST Service.</td>
</tr>
<tr>
<td>Vendor specific REST service GUID</td>
<td>6</td>
<td>16</td>
<td>GUID of vendor specific REST service.</td>
</tr>
<tr>
<td>Vendor defined data</td>
<td>22</td>
<td>n</td>
<td>Vendor-defined data.</td>
</tr>
</tbody>
</table>

Device path example of In-band vendor-specific REST Service through BMC:

\[
PciRoot(0x2)/Pci(0x2,0x0)/Pci(0x0,0x0)/BMC(0,0xf0000000)/RestService(0xff, 1, 00000000-0000-0000-0000-0000000000000000,0,0)
\]

### 10.3.5 Media Device Path

This Device Path is used to describe the portion of the medium that is being abstracted by a boot service. An example of Media Device Path would be defining which partition on a hard drive was being used.

#### 10.3.5.1 Hard Drive

The Hard Drive Media Device Path is used to represent a partition on a hard drive. Each partition has at least Hard Drive Device Path node, each describing an entry in a partition table. EFI supports MBR and GPT partitioning formats. Partitions are numbered according to their entry in their respective partition table, starting with 1. Partitions are addressed in EFI starting at LBA zero. A partition number of zero can be used to represent the raw hard drive or a raw extended partition.

The partition format is stored in the Device Path to allow new partition formats to be supported in the future. The Hard Drive Device Path also contains a Disk Signature and a Disk Signature Type. The disk signature is maintained by the OS and only used by EFI to partition Device Path nodes. The disk signature enables the OS to find disks even after they have been physically moved in a system.

Section 3.1.2 defines special rules for processing the Hard Drive Media Device Path. These special rules enable a disk’s location to change and still have the system boot from the disk.

### Table 10-49 Hard Drive Media Device Path

<table>
<thead>
<tr>
<th>Mnemonic</th>
<th>Byte Offset</th>
<th>Byte Length</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Type</td>
<td>0</td>
<td>1</td>
<td>Type 4 – Media Device Path</td>
</tr>
<tr>
<td>Sub-Type</td>
<td>1</td>
<td>1</td>
<td>Sub-Type 1 – Hard Drive</td>
</tr>
<tr>
<td>Length</td>
<td>2</td>
<td>2</td>
<td>Length of this structure in bytes. Length is 42 bytes.</td>
</tr>
</tbody>
</table>
10.3.5.2 CD-ROM Media Device Path

The CD-ROM Media Device Path is used to define a system partition that exists on a CD-ROM. The CD-ROM is assumed to contain an ISO-9660 file system and follow the CD-ROM “El Torito” format. The Boot Entry number from the Boot Catalog is how the “El Torito” specification defines the existence of bootable entities on a CD-ROM. In EFI the bootable entity is an EFI System Partition that is pointed to by the Boot Entry.

<table>
<thead>
<tr>
<th>Mnemonic</th>
<th>Byte Offset</th>
<th>Byte Length</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Partition Number</td>
<td>4</td>
<td>4</td>
<td>Describes the entry in a partition table, starting with entry 1. Partition number zero represents the entire device. Valid partition numbers for a MBR partition are [1, 4]. Valid partition numbers for a GPT partition are [1, NumberOfPartitionEntries].</td>
</tr>
<tr>
<td>Partition Start</td>
<td>8</td>
<td>8</td>
<td>Starting LBA of the partition on the hard drive</td>
</tr>
<tr>
<td>Partition Size</td>
<td>16</td>
<td>8</td>
<td>Size of the partition in units of Logical Blocks</td>
</tr>
<tr>
<td>Partition Signature</td>
<td>24</td>
<td>16</td>
<td>Signature unique to this partition: If SignatureType is 0, this field has to be initialized with 16 zeroes. If SignatureType is 1, the MBR signature is stored in the first 4 bytes of this field. The other 12 bytes are initialized with zeroes. If SignatureType is 2, this field contains a 16 byte signature.</td>
</tr>
<tr>
<td>Partition Format</td>
<td>40</td>
<td>1</td>
<td>Partition Format: (Unused values reserved) 0x01 – PC-AT compatible legacy MBR (see Section 5.2.1). Partition Start and Partition Size come from PartitionStartingLBA and PartitionSizeInLBA for the partition. 0x02 – GUID Partition Table (see Section 5.3.2).</td>
</tr>
<tr>
<td>Signature Type</td>
<td>41</td>
<td>1</td>
<td>Type of Disk Signature: (Unused values reserved) 0x00 – No Disk Signature. 0x01 – 32-bit signature from address 0x1b8 of the type 0x01 MBR. 0x02 – GUID signature.</td>
</tr>
</tbody>
</table>
Table 10-50 CD-ROM Media Device Path

<table>
<thead>
<tr>
<th>Mnemonic</th>
<th>Byte Offset</th>
<th>Byte Length</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Type</td>
<td>0</td>
<td>1</td>
<td>Type 4 – Media Device Path.</td>
</tr>
<tr>
<td>Sub-Type</td>
<td>1</td>
<td>1</td>
<td>Sub-Type 2 – CD-ROM “El Torito” Format.</td>
</tr>
<tr>
<td>Length</td>
<td>2</td>
<td>2</td>
<td>Length of this structure in bytes. Length is 24 bytes.</td>
</tr>
<tr>
<td>Boot Entry</td>
<td>4</td>
<td>4</td>
<td>Boot Entry number from the Boot Catalog. The Initial/Default entry is defined as zero.</td>
</tr>
<tr>
<td>Partition Start</td>
<td>8</td>
<td>8</td>
<td>Starting RBA of the partition on the medium. CD-ROMs use Relative logical Block Addressing.</td>
</tr>
<tr>
<td>Partition Size</td>
<td>16</td>
<td>8</td>
<td>Size of the partition in units of Blocks, also called Sectors.</td>
</tr>
</tbody>
</table>

10.3.5.3 Vendor-Defined Media Device Path

Table 10-51 Vendor-Defined Media Device Path

<table>
<thead>
<tr>
<th>Mnemonic</th>
<th>Byte Offset</th>
<th>Byte Length</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Type</td>
<td>0</td>
<td>1</td>
<td>Type 4 – Media Device Path.</td>
</tr>
<tr>
<td>Sub-Type</td>
<td>1</td>
<td>1</td>
<td>Sub-Type 3 – Vendor.</td>
</tr>
<tr>
<td>Length</td>
<td>2</td>
<td>2</td>
<td>Length of this structure in bytes. Length is 20 + n bytes.</td>
</tr>
<tr>
<td>Vendor GUID</td>
<td>4</td>
<td>16</td>
<td>Vendor-assigned GUID that defines the data that follows.</td>
</tr>
<tr>
<td>Vendor Defined Data</td>
<td>20</td>
<td>n</td>
<td>Vendor-defined variable size data.</td>
</tr>
</tbody>
</table>

10.3.5.4 File Path Media Device Path

Table 10-52 File Path Media Device Path

<table>
<thead>
<tr>
<th>Mnemonic</th>
<th>Byte Offset</th>
<th>Byte Length</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Type</td>
<td>0</td>
<td>1</td>
<td>Type 4 – Media Device Path.</td>
</tr>
<tr>
<td>Sub-Type</td>
<td>1</td>
<td>1</td>
<td>Sub-Type 4 – File Path.</td>
</tr>
<tr>
<td>Length</td>
<td>2</td>
<td>2</td>
<td>Length of this structure in bytes. Length is 4 + n bytes.</td>
</tr>
<tr>
<td>Path Name</td>
<td>4</td>
<td>N</td>
<td>A NULL-terminated Path string including directory and file names. The length of this string n can be determined by subtracting 4 from the Length entry. A device path may contain one or more of these nodes. Each node can optionally add a &quot;&quot; separator to the beginning and/or the end of the Path Name string. The complete path to a file can be found by logically concatenating all the Path Name strings in the File Path Media Device Path nodes. This is typically used to describe the directory path in one node, and the filename in another node.</td>
</tr>
</tbody>
</table>
Rules for Path Name conversion:

- When concatenating two Path Names, ensure that the resulting string does not contain a double-separator "\". If it does, convert that double-separator to a single-separator.
- In the case where a Path Name which has no end separator is being concatenated to a Path Name with no beginning separator, a separator will need to be inserted between the Path Names.
- Single file path nodes with no directory path data are presumed to have their files located in the root directory of the device.

10.3.5.5 Media Protocol Device Path

The Media Protocol Device Path is used to denote the protocol that is being used in a device path at the location of the path specified. Many protocols are inherent to the style of device path.

<table>
<thead>
<tr>
<th>Mnemonic</th>
<th>Byte Offset</th>
<th>Byte Length</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Type</td>
<td>0</td>
<td>1</td>
<td>Type 4 – Media Device Path.</td>
</tr>
<tr>
<td>Sub-Type</td>
<td>1</td>
<td>1</td>
<td>Sub-Type 5 – Media Protocol.</td>
</tr>
<tr>
<td>Length</td>
<td>2</td>
<td>2</td>
<td>Length of this structure in bytes. Length is 20 bytes.</td>
</tr>
<tr>
<td>Protocol GUID</td>
<td>4</td>
<td>16</td>
<td>The ID of the protocol.</td>
</tr>
</tbody>
</table>

10.3.5.6 PIWG Firmware File

This type is used by systems implementing the UEFI PI Specification to describe a firmware file. The exact format and usage are defined in that specification.

<table>
<thead>
<tr>
<th>Mnemonic</th>
<th>Byte Offset</th>
<th>Byte Length</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Type</td>
<td>0</td>
<td>1</td>
<td>Type 4 – Media Device Path.</td>
</tr>
<tr>
<td>Sub-Type</td>
<td>1</td>
<td>1</td>
<td>Sub-Type 6 – PIWG Firmware File.</td>
</tr>
<tr>
<td>Length</td>
<td>2</td>
<td>2</td>
<td>Length of this structure in bytes. Length is 4 + n bytes.</td>
</tr>
<tr>
<td>...</td>
<td>4</td>
<td>n</td>
<td>Contents are defined in the UEFI PI Specification.</td>
</tr>
</tbody>
</table>
10.3.5.7 PIWG Firmware Volume

This type is used by systems implementing the UEFI PI Specification to describe a firmware volume. The exact format and usage are defined in that specification.

Table 10-55 PIWG Firmware Volume Device Path

<table>
<thead>
<tr>
<th>Mnemonic</th>
<th>Byte Offset</th>
<th>Byte Length</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Type</td>
<td>0</td>
<td>1</td>
<td>Type 4 – Media Device Path</td>
</tr>
<tr>
<td>Sub-Type</td>
<td>1</td>
<td>1</td>
<td>Sub-Type 7 – PIWG Firmware Volume</td>
</tr>
<tr>
<td>Length</td>
<td>2</td>
<td>2</td>
<td>Length of this structure in bytes. Length is 4 + n bytes.</td>
</tr>
<tr>
<td>...</td>
<td>4</td>
<td>n</td>
<td>Contents are defined in the UEFI PI Specification.</td>
</tr>
</tbody>
</table>

10.3.5.8 Relative Offset Range

This device path node specifies a range of offsets relative to the first byte available on the device. The starting offset is the first byte of the range and the ending offset is the last byte of the range (not the last byte + 1).

Table 10-56 Relative Offset Range

<table>
<thead>
<tr>
<th>Mnemonic</th>
<th>Byte Offset</th>
<th>Byte Length</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Type</td>
<td>0</td>
<td>1</td>
<td>Type 4 – Media Device Path</td>
</tr>
<tr>
<td>Sub-Type</td>
<td>1</td>
<td>1</td>
<td>Sub-Type 8 – Relative Offset Range</td>
</tr>
<tr>
<td>Length</td>
<td>2</td>
<td>2</td>
<td>Length of this structure in bytes.</td>
</tr>
<tr>
<td>Reserved</td>
<td>4</td>
<td>4</td>
<td>Reserved for future use.</td>
</tr>
<tr>
<td>Starting Offset</td>
<td>8</td>
<td>8</td>
<td>Offset of the first byte, relative to the parent device node.</td>
</tr>
<tr>
<td>Ending Offset</td>
<td>16</td>
<td>8</td>
<td>Offset of the last byte, relative to the parent device node.</td>
</tr>
</tbody>
</table>

10.3.5.9 RAM Disk

Table 10-57 RAM Disk Device Path

<table>
<thead>
<tr>
<th>Mnemonic</th>
<th>Byte Offset</th>
<th>Byte Length</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Type</td>
<td>0</td>
<td>1</td>
<td>Type 4 – Media Device Path</td>
</tr>
<tr>
<td>Sub-Type</td>
<td>1</td>
<td>1</td>
<td>Sub Type 9 – RAM Disk Device Path</td>
</tr>
<tr>
<td>Length</td>
<td>2</td>
<td>2</td>
<td>Length of this structure in bytes. Length is 38 bytes.</td>
</tr>
<tr>
<td>Starting Address</td>
<td>4</td>
<td>8</td>
<td>Starting Memory Address.</td>
</tr>
<tr>
<td>Ending Address</td>
<td>12</td>
<td>8</td>
<td>Ending Memory Address.</td>
</tr>
<tr>
<td>Disk Type GUID</td>
<td>20</td>
<td>16</td>
<td>GUID that defines the type of the RAM Disk. The GUID can be any of the values defined below, or a vendor defined GUID.</td>
</tr>
<tr>
<td>Disk Instance</td>
<td>36</td>
<td>2</td>
<td>RAM Disk instance number, if supported. The default value is zero.</td>
</tr>
</tbody>
</table>
The following GUIDs are used with a RAM Disk Device Path to describe the RAM Disk Type. Additional GUIDs can be generated to describe additional RAM Disk Types. The Disk Type GUID values used in the RAM Disk device path must match the corresponding values in the Address Range Type GUID of the ACPI NFIT table. Refer to the ACPI specification for details.

This GUID defines a RAM Disk supporting a raw disk format in volatile memory:

```
#define EFI_VIRTUAL_DISK_GUID \
{ 0x77AB535A,0x45FC,0x624B,\ 
{0x55,0x60,0xF7,0x82,0x81,0xD1,0xF9,0x6E } }
```

This GUID defines a RAM Disk supporting an ISO image in volatile memory:

```
#define EFI_VIRTUAL_CD_GUID \
{ 0x3D5ABD30,0x4175,0x87CE,\ 
{0x6D,0x64,0x52,0xAD,0xE5,0x23,0xC4,0xBB } }
```

This GUID defines a RAM Disk supporting a raw disk format in persistent memory:

```
#define EFI_PERSISTENT_VIRTUAL_DISK_GUID \
{ 0x5CEA02C9,0x4D07,0x69D3,\ 
{0x26,0x9F,0x44,0x96,0xFB,0xE0,0x96,0xF9 } }
```

This GUID defines a RAM Disk supporting an ISO image in persistent memory:

```
#define EFI_PERSISTENT_VIRTUAL_CD_GUID \
{ 0x08018188,0x42CD,0xBB48,\ 
{0x10,0x0F,0x53,0x87,0xD5,0x3D,0xED,0x3D } }
```

10.3.6 BIOS Boot Specification Device Path

This Device Path is used to describe the booting of non-EFI-aware operating systems. This Device Path is based on the IPL and BCV table entry data structures defined in Appendix A of the BIOS Boot Specification. The BIOS Boot Specification Device Path defines a complete Device Path and is not used with other Device Path entries. This Device Path is only needed to enable platform firmware to select a legacy non-EFI OS as a boot option.
Table 10-58 BIOS Boot Specification Device Path

<table>
<thead>
<tr>
<th>Mnemonic</th>
<th>Byte Offset</th>
<th>Byte Length</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Type</td>
<td>0</td>
<td>1</td>
<td>Type 5 – BIOS Boot Specification Device Path.</td>
</tr>
<tr>
<td>Sub-Type</td>
<td>1</td>
<td>1</td>
<td>Sub-Type 1 – BIOS Boot Specification Version 1.01.</td>
</tr>
<tr>
<td>Length</td>
<td>2</td>
<td>2</td>
<td>Length of this structure in bytes. Length is 8 + n bytes.</td>
</tr>
<tr>
<td>Device Type</td>
<td>4</td>
<td>2</td>
<td>Device Type as defined by the BIOS Boot Specification.</td>
</tr>
<tr>
<td>Status Flag</td>
<td>6</td>
<td>2</td>
<td>Status Flags as defined by the BIOS Boot Specification.</td>
</tr>
<tr>
<td>Description String</td>
<td>8</td>
<td>n</td>
<td>A null-terminated ASCII string that describes the boot device to a user. The size of this string n can be determined by subtracting 8 from the Length entry.</td>
</tr>
</tbody>
</table>

Example BIOS Boot Specification Device Types include:

- 00h = Reserved
- 01h = Floppy
- 02h = Hard Disk
- 03h = CD-ROM
- 04h = PCMCIA
- 05h = USB
- 06h = Embedded network
- 07h..7Fh = Reserved
- 80h = BEV device
- 81h..FEh = Reserved
- FFh = Unknown

**Note:** When UEFI Secure Boot is enabled, attempts to boot non-UEFI OS shall fail; see Section 32.4.

### 10.4 Device Path Generation Rules

#### 10.4.1 Housekeeping Rules

The Device Path is a set of Device Path nodes. The Device Path must be terminated by an End of Device Path node with a sub-type of End the Entire Device Path. A NULL Device Path consists of a single End Device Path Node. A Device Path that contains a NULL pointer and no Device Path structures is illegal.

All Device Path nodes start with the generic Device Path structure. Unknown Device Path types can be skipped when parsing the Device Path since the length field can be used to find the next Device Path structure in the stream. Any future additions to the Device Path structure types will always start with the current standard header. The size of a Device Path can be determined by traversing the generic Device Path structures in each header and adding up the total size of the Device Path. This size will include the four bytes of the End of Device Path structure.
Multiple hardware devices may be pointed to by a single Device Path. Each hardware device will contain a complete Device Path that is terminated by the Device Path End Structure. The Device Path End Structures that do not end the Device Path contain a sub-type of End This Instance of the Device Path. The last Device Path End Structure contains a sub-type of End Entire Device Path.

10.4.2 Rules with ACPI _HID and _UID

As described in the ACPI specification, ACPI supports several different kinds of device identification objects, including _HID, _CID and _UID. The _UID device identification objects are optional in ACPI and only required if more than one _HID exists with the same ID. The ACPI Device Path structure must contain a zero in the _UID field if the ACPI name space does not implement _UID. The _UID field is a unique serial number that persists across reboots.

If a device in the ACPI name space has a _HID and is described by a _CRS (Current Resource Setting) then it should be described by an ACPI Device Path structure. A _CRS implies that a device is not mapped by any other standard. A _CRS is used by ACPI to make a nonstandard device into a Plug and Play device. The configuration methods in the ACPI name space allow the ACPI driver to configure the device in a standard fashion. The presence of a _CID determines whether the ACPI Device Path node or the Expanded ACPI Device Path node should be used.

Table 10-59 maps ACPI _CRS devices to EFI Device Path.

<table>
<thead>
<tr>
<th>ACPI _CRS Item</th>
<th>EFI Device Path</th>
</tr>
</thead>
<tbody>
<tr>
<td>PCI Root Bus</td>
<td>ACPI Device Path: _HID PNP0A03, _UID</td>
</tr>
<tr>
<td>Floppy</td>
<td>ACPI Device Path: _HID PNP0604, _UID drive select encoding 0-3</td>
</tr>
<tr>
<td>Keyboard</td>
<td>ACPI Device Path: _HID PNP0301, _UID 0</td>
</tr>
<tr>
<td>Serial Port</td>
<td>ACPI Device Path: _HID PNP0501, _UID Serial Port COM number 0-3</td>
</tr>
<tr>
<td>Parallel Port</td>
<td>ACPI Device Path: _HID PNP0401, _UID LPT number 0-3</td>
</tr>
</tbody>
</table>

Support of root PCI bridges requires special rules in the EFI Device Path. A root PCI bridge is a PCI device usually contained in a chipset that consumes a proprietary bus and produces a PCI bus. In typical desktop and mobile systems there is only one root PCI bridge. On larger server systems there are typically multiple root PCI bridges. The operation of root PCI bridges is not defined in any current PCI specification. A root PCI bridge should not be confused with a PCI to PCI bridge that both consumes and produces a PCI bus. The operation and configuration of PCI to PCI bridges is fully specified in current PCI specifications.

Root PCI bridges will use the plug and play ID of PNP0A03, This will be stored in the ACPI Device Path _HID field, or in the Expanded ACPI Device Path _CID field to match the ACPI name space. The _UID in the ACPI Device Path structure must match the _UID in the ACPI name space.

10.4.3 Rules with ACPI _ADR

If a device in the ACPI name space can be completely described by a _ADR object then it will map to an EFI ACPI, Hardware, or Message Device Path structure. A _ADR method implies a bus with a standard enumeration algorithm. If the ACPI device has a _ADR and a _CRS method, then it should also have a _HID method and follow the rules for using _HID.
Table 10-60 relates the ACPI _ADR bus definition to the EFI Device Path:

<table>
<thead>
<tr>
<th>ACPI _ADR Bus</th>
<th>EFI Device Path</th>
</tr>
</thead>
<tbody>
<tr>
<td>EISA</td>
<td>Not supported</td>
</tr>
<tr>
<td>Floppy Bus</td>
<td>ACPI Device Path: _HID PNP0604, _UID drive select encoding 0-3</td>
</tr>
<tr>
<td>IDE Controller</td>
<td>ATAPI Message Device Path: Maser/Slave : LUN</td>
</tr>
<tr>
<td>IDE Channel</td>
<td>ATAPI Message Device Path: Maser/Slave : LUN</td>
</tr>
<tr>
<td>PCI</td>
<td>PCI Hardware Device Path</td>
</tr>
<tr>
<td>PCMCIA</td>
<td>Not Supported</td>
</tr>
<tr>
<td>PC CARD</td>
<td>PC CARD Hardware Device Path</td>
</tr>
<tr>
<td>SMBus</td>
<td>Not Supported</td>
</tr>
<tr>
<td>SATA bus</td>
<td>SATA Messaging Device Path</td>
</tr>
</tbody>
</table>

### 10.4.4 Hardware vs. Messaging Device Path Rules

Hardware Device Paths are used to define paths on buses that have a standard enumeration algorithm and that relate directly to the coherency domain of the system. The coherency domain is defined as a global set of resources that is visible to at least one processor in the system. In a typical system this would include the processor memory space, IO space, and PCI configuration space.

Messaging Device Paths are used to define paths on buses that have a standard enumeration algorithm, but are not part of the global coherency domain of the system. SCSI and Fibre Channel are examples of this kind of bus. The Messaging Device Path can also be used to describe virtual connections over network-style devices. An example would be the TCP/IP address of an internet connection.

Thus Hardware Device Path is used if the bus produces resources that show up in the coherency resource domain of the system. A Message Device Path is used if the bus consumes resources from the coherency domain and produces resources out side the coherency domain of the system.

### 10.4.5 Media Device Path Rules

The Media Device Path is used to define the location of information on a medium. Hard Drives are subdivided into partitions by the MBR and a Media Device Path is used to define which partition is being used. A CD-ROM has boot partitions that are defined by the “El Torito” specification, and the Media Device Path is used to point to these partitions.

An EFI_BLOCK_IO_PROTOCOL is produced for both raw devices and partitions on devices. This allows the EFI_SIMPLE_FILE_SYSTEM_PROTOCOL protocol to not have to understand media formats. The EFI_BLOCK_IO_PROTOCOL for a partition contains the same Device Path as the parent EFI_BLOCK_IO_PROTOCOL for the raw device with the addition of a Media Device Path that defines which partition is being abstracted.

The Media Device Path is also used to define the location of a file in a file system. This Device Path is used to load files and to represent what file an image was loaded from.
10.4.6 Other Rules

The BIOS Boot Specification Device Path is not a typical Device Path. A Device Path containing the BIOS Boot Specification Device Path should only contain the required End Device Path structure and no other Device Path structures. The BIOS Boot Specification Device Path is only used to allow the EFI boot menus to boot a legacy operating system from legacy media.

The EFI Device Path can be extended in a compatible fashion by assigning your own vendor GUID to a Hardware, Messaging, or Media Device Path. This extension is guaranteed to never conflict with future extensions of this specification.

The EFI specification reserves all undefined Device Path types and subtypes. Extension is only permitted using a Vendor GUID Device Path entry.

10.5 Device Path Utilities Protocol

This section describes the `EFI_DEVICE_PATH_UTILS_PROTOCOL`, which aids in creating and manipulating device paths.

**EFI_DEVICE_PATH_UTILS_PROTOCOL**

**Summary**

Creates and manipulates device paths and device nodes.

**GUID**

```c
// {0379BE4E-D706-437d-B037-EDB82FB772A4}
#define EFI_DEVICE_PATH_UTILS_PROTOCOL_GUID 
{0x379be4e,0xd706,0x437d,0xb0,0x37,0xed,0xb8,0x2f,0xb7,0x72,0xa4 }
```

**Protocol Interface Structure**

```c
typedef struct _EFI_DEVICE_PATH_UTILS_PROTOCOL {
    EFI_DEVICE_PATH_UTILS_GET_DEVICE_PATH_SIZE  GetDevicePathSize;
    EFI_DEVICE_PATH_UTILS_DUP_DEVICE_PATH      DuplicateDevicePath;
    EFI_DEVICE_PATH_UTILS_APPEND_PATH          AppendDevicePath;
    EFI_DEVICE_PATH_UTILS_APPEND_NODE          AppendDeviceNode;
    EFI_DEVICE_PATH_UTILS_APPEND_INSTANCE      AppendDevicePathInstance;
    EFI_DEVICE_PATH_UTILS_GET_NEXT_INSTANCE   GetNextDevicePathInstance;
    EFI_DEVICE_PATH_UTILS_IS_MULTI_INSTANCE    IsDevicePathMultiInstance;
    EFI_DEVICE_PATH_UTILS_CREATE_NODE         CreateDeviceNode;
} EFI_DEVICE_PATH_UTILS_PROTOCOL;
```

**Parameters**

- `GetDevicePathSize` Returns the size of the specified device path, in bytes.
- `DuplicateDevicePath` Duplicates a device path structure.
- `AppendDeviceNode` Appends the device node to the specified device path.
- `AppendDevicePath` Appends the device path to the specified device path.
AppendDevicePathInstance

Appends a device path instance to another device path.

GetNextDevicePathInstance

Retrieves the next device path instance from a device path data structure.

IsDevicePathMultiInstance

Returns TRUE if this is a multi-instance device path.

CreateDeviceNode

Allocates memory for a device node with the specified type and subtype.

Description

The **EFI DEVICE PATH UTILITIES_PROTOCOL** provides common utilities for creating a manipulating device paths and device nodes.

EFI_DEVICE_PATH_UTILITIES_PROTOCOL.GetDevicePathSize()

**Summary**

Returns the size of the device path, in bytes.

Prototype

typedef

UINTN

(EIFIAPIM *EFI_DEVICE_PATH_UTILS_GET_DEVICE_PATH_SIZE) (  
    IN CONST EFI_DEVICE_PATH_PROTOCOL *DevicePath  
);  

Parameters

DevicePath  

Points to the start of the EFI device path.

Description

This function returns the size of the specified device path, in bytes, including the end-of-path tag. If DevicePath is **NULL** then zero is returned.

Related Definitions

**EFI_DEVICE_PATH_PROTOCOL** is defined in Section 10.2.

EFI_DEVICE_PATH_UTILITIES_PROTOCOL.DuplicateDevicePath()

**Summary**

Create a duplicate of the specified path.

Prototype

typedef

EFI DEVICE PATH_PROTOCOL*

(EIFIAPIM *EFI_DEVICE_PATH_UTILS_DUP_DEVICE_PATH)(  
    IN CONST EFI DEVICE_PATH_PROTOCOL *DevicePath  
);  

Parameters

DevicePath  

Points to the start of the EFI device path.
Parameters
DevicePath  Points to the source device path.

Description
This function creates a duplicate of the specified device path. The memory is allocated from EFI boot services memory. It is the responsibility of the caller to free the memory allocated. If DevicePath is NULL then NULL will be returned and no memory will be allocated.

Related Definitions
EFI_DEVICE_PATH_PROTOCOL is defined in Section 10.2.

Returns
This function returns a pointer to the duplicate device path or NULL if there was insufficient memory.

EFI_DEVICE_PATH_UTILITIES_PROTOCOL.AppendDevicePath()

Summary
Create a new path by appending the second device path to the first.

Prototype
typedef
EFI_DEVICE_PATH_PROTOCOL*
(EIFIAPI *EFI_DEVICE_PATH_UTILS_APPEND_PATH) (  
IN CONST EFI_DEVICE_PATH_PROTOCOL *Src1,  
IN CONST EFI_DEVICE_PATH_PROTOCOL *Src2  
);

Parameters
Src1  Points to the first device path.  
Src2  Points to the second device path.

Description
This function creates a new device path by appending a copy of the second device path to a copy of the first device path in a newly allocated buffer. Only the end-of-device-path device node from the second device path is retained. If Src1 is NULL and Src2 is non-NULL, then a duplicate of Src2 is returned. If Src1 is non-NULL and Src2 is NULL, then a duplicate of Src1 is returned. If Src1 and Src2 are both NULL, then a copy of an end-of-device-path is returned.

The memory is allocated from EFI boot services memory. It is the responsibility of the caller to free the memory allocated.

Related Definitions
EFI_DEVICE_PATH_PROTOCOL is defined in Section 10.2.
Returns
This function returns a pointer to the newly created device path or **NULL** if memory could not be allocate.

`EFI_DEVICE_PATH_UTILITIES_PROTOCOL.AppendDeviceNode()`

**Summary**
Creates a new path by appending the device node to the device path.

**Prototype**
```c
typedef EFI_DEVICE_PATH_PROTOCOL* (EFIAPI *EFI_DEVICE_PATH_UTILS_APPEND_NODE) (
    IN CONST EFI_DEVICE_PATH_PROTOCOL *DevicePath,
    IN CONST EFI_DEVICE_PATH_PROTOCOL *DeviceNode
);
```

**Parameters**
- **DevicePath** Points to the device path.
- **DeviceNode** Points to the device node.

**Description**
This function creates a new device path by appending a copy of the specified device node to a copy of the specified device path in an allocated buffer. The end-of-device-path device node is moved after the end of the appended device node. If **DeviceNode** is **NULL** then a copy of **DevicePath** is returned. If **DevicePath** is **NULL** then a copy of **DeviceNode**, followed by an end-of-device path device node is returned. If both **DeviceNode** and **DevicePath** are **NULL** then a copy of an end-of-device-path device node is returned.

The memory is allocated from EFI boot services memory. It is the responsibility of the caller to free the memory allocated.

**Related Definitions**
`EFI_DEVICE_PATH_PROTOCOL` is defined in Section 10.2.

**Returns**
This function returns a pointer to the allocated device path, or **NULL** if there was insufficient memory.

`EFI_DEVICE_PATH_UTILITIES_PROTOCOL.AppendDevicePathInstance()`

**Summary**
Creates a new path by appending the specified device path instance to the specified device path.
Prototype

typedef

EFI_DEVICE_PATH_PROTOCOL*

(EIFIAPI *EFI_DEVICE_PATH_UTILS_APPEND_INSTANCE) (

IN CONST EFI_DEVICE_PATH_PROTOCOL *DevicePath,

IN CONST EFI_DEVICE_PATH_PROTOCOL *DevicePathInstance

);

Parameters

DevicePathPoints to the device path. If NULL, then ignored.

DevicePathInstancePoints to the device path instance

Description

This function creates a new device path by appending a copy of the specified device path instance to a copy of the specified device path in an allocated buffer. The end-of-device-path device node is moved after the end of the appended device node and a new end-of-device-path-instance node is inserted between. If DevicePath is NULL, then a copy if DevicePathInstance is returned instead.

The memory is allocated from EFI boot services memory. It is the responsibility of the caller to free the memory allocated.

Related Definitions

EFI_DEVICE_PATH_PROTOCOL is defined in Section 10.2.

Returns

This function returns a pointer to the newly created device path or NULL if DevicePathInstance is NULL or there was insufficient memory.

EFI_DEVICE_PATH_UTILITIES_PROTOCOL.GetNextDevicePathInstance()

Summary

Creates a copy of the current device path instance and returns a pointer to the next device path instance.

Prototype

typedef

EFI_DEVICE_PATH_PROTOCOL*

(EIFIAPI *EFIDEVICE_PATH_UTILITIES_GET_NEXT_INSTANCE) (

IN OUT EFI_DEVICE_PATH_PROTOCOL **DevicePathInstance,

OUT UINTN *DevicePathInstanceSize OPTIONAL

);

Parameters

DevicePathInstanceOn input, this holds the pointer to the current device path instance.

On output, this holds the pointer to the next device path instance or NULL if there are no more device path instances in the device path.
**DevicePathInstanceSize**

On output, this holds the size of the device path instance, in bytes or zero, if DevicePathInstance is **NULL**. If **NULL**, then the instance size is not output.

**Description**

This function creates a copy of the current device path instance. It also updates DevicePathInstance to point to the next device path instance in the device path (or **NULL** if no more) and updates DevicePathInstanceSize to hold the size of the device path instance copy.

The memory is allocated from EFI boot services memory. It is the responsibility of the caller to free the memory allocated.

**Related Definitions**

**EFI_DEVICE_PATH_PROTOCOL** is defined in Section 10.2.

**Returns**

This function returns a pointer to the copy of the current device path instance or **NULL** if DevicePathInstance was **NULL** on entry or there was insufficient memory.

**EFI_DEVICE_PATH_UTILITIES_PROTOCOL.CreateDeviceNode()**

**Summary**

Creates a device node

**Prototype**

```c
typedef EFIDEVICEPATHPROTOCOL* (EFIAPI*EFI_DEVICE_PATH_UTILS_CREATE_NODE) (IN UINT8 NodeType,
                         IN UINT8 NodeSubType,
                         IN UINT16 NodeLength);
```

**Parameters**

- **NodeType**
  - NodeType is the device node type (**EFI_DEVICE_PATH_PROTOCOL.Type**) for the new device node.

- **NodeSubType**
  - NodeSubType is the device node sub-type (**EFI_DEVICE_PATH_PROTOCOL.SubType**) for the new device node.

- **NodeLength**
  - NodeLength is the length of the device node (**EFI_DEVICE_PATH_PROTOCOL.Length**) for the new device node.

**Description**

This function creates a new device node in a newly allocated buffer.
The memory is allocated from EFI boot services memory. It is the responsibility of the caller to free the memory allocated.

**Related Definitions**

`EFI_DEVICE_PATH_PROTOCOL` is defined in [Section 10.2](#).

**Returns**

This function returns a pointer to the created device node or NULL if `NodeLength` is less than the size of the header or there was insufficient memory.

**`EFI DEVICE PATH UTILITIES_PROTOCOL.IsDevicePathMultiInstance()`**

**Summary**

Returns whether a device path is multi-instance.

**Prototype**

```c
typedef BOOLEAN (EFIAPI *EFI DEVICE PATH UTILS IS MULTI INSTANCE) (
    IN CONST EFI DEVICE PATH_PROTOCOL *DevicePath
);
```

**Parameters**

- `DevicePath` Points to the device path. If `NULL`, then ignored.

**Description**

This function returns whether the specified device path has multiple path instances.

**Related Definitions**

`EFI_DEVICE_PATH_PROTOCOL` is defined in [Section 10.2](#).

**Returns**

This function returns TRUE if the device path has more than one instance or FALSE if it is empty or contains only a single instance.

### 10.6 EFI Device Path Display Format Overview

This section describes the recommended conversion between an EFI Device Path Protocol and text. It also describes standard protocols for implementing these. The goals are:

- Standardized display format. This allows documentation and test tools to understand output coming from drivers provided by multiple vendors.
- Increase Readability. Device paths need to be read by people, so the format should be in a form which can be deciphered, maintaining as much as possible the industry standard means of presenting data. In this case, there are two forms, a display-only form and a parse-able form.
- Round-trip conversion from text to binary form and back to text without loss, if desired.
• Ease of command-line parsing. Since device paths can appear on the command-lines of UEFI applications executed from a shell, the conversion format should not prohibit basic command-line processing, either by the application or by a shell.

10.6.1 Design Discussion

The following subsections describe the design considerations for conversion to and from the EFI Device Path Protocol binary format and its corresponding text form.

10.6.1.1 Standardized Display Format

Before the UEFI 2.0, there was no standardized format for the conversion from the EFI Device Path protocol and text. Some de-facto standards arose, either as part of the standard implementation or in descriptive text in the EFI Device Driver Writer’s Guide, although they didn’t agree. The standardized format attempts to maintain at least the spirit of these earlier ideas.

10.6.1.2 Readability

Since these are conversions to text and, in many cases, users have to read and understand the text form of the EFI Device Path, it makes sense to make them as readable as reasonably possible. Several strategies are used to accomplish this:

• Creating simplified forms for well-known device paths. For example, a PCI root Bridge can be represented as Acpi(PNP0A03,0), but makes more sense as PciRoot(0). When converting from text to binary form, either form is accepted, but when converting from binary form to text, the latter is preferred.

• Omitting the conversion of fields which have empty or default values. By doing this, the average display length is greatly shortened, which improves readability.

10.6.1.3 Round-Trip Conversion

The conversions specified here guarantee at least that conversion to and from the binary representation of the EFI Device Path will be semantically identical.

\[
\text{Text}_1 \xrightarrow{\delta} \text{Binary}_1 \xrightarrow{\delta} \text{Text}_2 \xrightarrow{\delta} \text{Binary}_2
\]

Figure 10-1 Text to Binary Conversion

In Figure 10-1, the process described in this section is applied to Text1, converting it to Binary1. Subsequently, Binary1 is converted to Text2. Finally, the Text2 is converted to Binary2. In these cases, Binary1 and Binary2 will always be identical. Text1 and Text2 may or may not be identical. This is the result of the fact that the text representation has, in some cases, more than one way of representing the same EFI Device Path node.

\[
\text{Binary}_1 \xrightarrow{\delta} \text{Text}_1 \xrightarrow{\delta} \text{Binary}_2 \xrightarrow{\delta} \text{Text}_2
\]

Figure 10-2 Binary to Text Conversion
In Figure 10-2 the process described in this section is applied to Binary1, converting it to Text1. Subsequently, Text1 is converted to Binary2. Finally, Binary2 is converted to Text2. In these cases, Binary1 and Binary2 will always be identical and Text1 and Text2 will always be identical.

Another consideration in round-trip conversion is potential ambiguity in parsing. This happens when the text representation could be converted into more than type of device node, thus requiring information beyond that contained in the text representation in order to determine the correct conversion to apply. In the case of EFI Device Paths, this causes problems primarily with literal strings in the device path, such as those found in file names, volumes or directories.

For example, the file name Acpi(PNP0A03,0) might be a legal FAT32 file name. However, in parsing this, it is not clear whether it refers to an Acpi device node or a file name. Thus, it is ambiguous. In order to prevent ambiguity, certain characters may only be used for device node keywords and may not be used in file names or directories.

10.6.1.4 Command-Line Parsing

Applications written to this specification need to accept the text representation of EFI device paths as command-line parameters, possibly in the context of a command-prompt or shell. In order to do this, the text representation must follow simple guidelines concerning its format.

Command-line parsing generally involves three separate concepts: substitution, redirection and division.

In substitution, the invoker of the application modifies the actual contents of the command-line before it is passed to the application. For example:

```plaintext
copy *.xyz
```

In redirection, the invoker of the application gleans from the command line parameters which it uses to, for example, redirect or pipe input or output. For example:

```plaintext
echo This text is copied to a file >abc

dir | more
```

Finally, in division, the invoker or the application startup code divides the command-line up into individual arguments. The following line, for example, has (at least) three arguments, divided by whitespace.

```plaintext
copy /b file1.info file2.info
```

10.6.1.5 Text Representation Basics

This section describes the basic rules for the text representation of device nodes and device paths. The formal grammar describing appears later.

The text representation of a device path (or text device path) consists of one or more text device nodes, each preceded by a ‘/’ or ‘\’ character. The behavior of a device path where the first node is not preceded by one of these characters is undefined. Some implementations may treat it as a relative path from a current working directory.

Spaces are not allowed at any point within the device path except when quoted with double quotes ("). The ‘|’ (bar), ‘<’ (less than) and ‘>’ (greater than) characters are likewise reserved for use by the shell.
There are two types of text device nodes: file-name/directory or canonical. Canonical text device nodes are prefixed by an option name consisting of only alphanumerical characters, followed by a parenthesis, followed by option-specific parameters separated by a ',' (comma). File names and directories have no prefixes.

Consider:

AcpiEx(HWP0002, PNP0A03,0)

Which could also be written:

AcpiEx(HWP0002,CID=PNP0A03) or

AcpiEx(HWP0002,PNP0A03)

Since CID and UID are optional parameters. Or consider:

Acpi(HWP0002,0)

Which could also be written:

Acpi(HWP0002)

Since UID is an optional parameter.
**Figure 10-5 Device Node Option Names**

### 10.6.1.6 Text Device Node Reference

In each of the following table rows, a specific device node type and sub-type are given, along with the most general form of the text representation. Any parameters for the device node are listed in italics. In each case, the type is listed and along with it what is required or optional, and any default value, if applicable.

On subsequent lines, alternate representations are listed. In general, these alternate representations are simplified by the assumption that one or more of the parameters is set to a specific value.

#### Parameter Types

This section describes the various types of option parameter values.

**Table 10-61 EFI Device Path Option Parameter Values**

<table>
<thead>
<tr>
<th>Parameter Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>GUID</td>
<td>An EFI GUID in standard format xxxxxxxx-xxxx-xxxx-xxxx-xxxxxxxxxxxx. See “GUID and Time Formats” on page 2178.</td>
</tr>
<tr>
<td>Keyword</td>
<td>In some cases, one of a series of keywords must be listed.</td>
</tr>
<tr>
<td>Integer</td>
<td>Unless otherwise specified, this indicates an unsigned integer in the range of 0 to 2^32-1. The value is decimal, unless preceded by “0x” or “0X”, in which case it is hexadecimal.</td>
</tr>
<tr>
<td>EISAID</td>
<td>A seven character text identifier in the format used by the ACPI specification. The first three characters must be alphabetic, either upper or lower case. The second four characters are hexadecimal digits, either numeric, upper case or lower case. Optionally, it can be the number 0.</td>
</tr>
<tr>
<td>String</td>
<td>Series of alphabetic, numeric and punctuation characters not including a right parenthesis ‘)’, bar ‘</td>
</tr>
<tr>
<td>HexDump</td>
<td>Series of bytes, represented by two hexadecimal characters per byte. Unless otherwise indicated, the size is only limited by the length of the device node.</td>
</tr>
<tr>
<td>IPv4 Address</td>
<td>Series of four integer values (each between 0-255), separated by a ‘.’ Optionally, followed by a ‘:’ and an integer value between 0-65555. If the ‘:’ is not present, then the port value is zero.</td>
</tr>
<tr>
<td>IPv6 Address</td>
<td>IPv6 Address is expressed in the format [address]:port. The ‘address’ is expressed in the way defined in RFC4291 Section 2.2. The ‘port’ after the [address] is optional. If present, the ‘port’ is an integer value between 0-65535 to represent the port number, or else, port number is zero.</td>
</tr>
</tbody>
</table>
### Table 10-62 Device Node Table

<table>
<thead>
<tr>
<th>Device Node Type/SubType/Other</th>
<th>Description</th>
</tr>
</thead>
</table>
| (when type is not recognized) | Path\((\text{type, subtype, data})\)  
The type is an integer from 0-255.  
The sub-type is an integer from 0-255.  
The data is a hex dump. |
| Type: 1 (Hardware Device Path) | HardwarePath\((\text{subtype, data})\)  
The subtype is an integer from 0-255.  
The data is a hex dump. |
| SubType: 1 (PCI) | Pci\((\text{Device, Function})\)  
The Device is an integer from 0-31 and is required.  
The Function is an integer from 0-7 and is required. |
| SubType: 2 (PcPcard) | PcCard\((\text{Function})\)  
The Function is an integer from 0-255 and is required. |
| SubType: 3 (Memory Mapped) | MemoryMapped\((\text{EfiMemoryType,StartingAddress, EndingAddress})\)  
The EfiMemoryType is a 32-bit integer and is required.  
The StartingAddress and EndingAddress are both 64-bit integers and are both required. |
| SubType: 4 (Vendor) | VenHw\((\text{Guid, Data})\)  
The Guid is a GUID and is required.  
The Data is a Hex Dump and is optional. The default value is zero bytes. |
| SubType: 5 (Controller) | Ctrl\((\text{Controller})\)  
The Controller is an integer and is required. |
| SubType: 6 (BMC) | BMC\((\text{Type,Address})\)  
The Type is an integer from 0-255, and is required.  
The Address is an unsigned 64-bit integer, and is required. |
| (when subtype is not recognized) | AcpiPath\((\text{subtype, data})\)  
The subtype is an integer from 0-255.  
The data is a hex dump. |
| SubType: 1 (ACPI Device Path) | Acpi\((\text{HID,UID})\)  
The HID parameter is an EISAID and is required.  
The UID parameter is an integer and is optional. The default value is zero. |
| SubType: 2 (ACPI Device Path) | PciRoot\((\text{UID})\)  
The UID parameter is an integer. It is optional but required for display. The default value is zero. |
| SubType: 1 (ACPI Device Path) | PcieRoot\((\text{UID})\)  
The UID parameter is an integer. It is optional but required for display. The default value is zero. |
<table>
<thead>
<tr>
<th>Device Node Type/SubType/Other</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Type: 2 (ACPI Device Path)</td>
<td>Floppy(UID)</td>
</tr>
<tr>
<td>SubType: 1 (ACPI Device Path)</td>
<td></td>
</tr>
<tr>
<td>HID=PNP0604</td>
<td></td>
</tr>
<tr>
<td></td>
<td>The UID parameter is an integer. It is optional for input but required for display. The default value is zero.</td>
</tr>
<tr>
<td>Type: 2 (ACPI Device Path)</td>
<td>Keyboard(UID)</td>
</tr>
<tr>
<td>SubType: 1 (ACPI Device Path)</td>
<td></td>
</tr>
<tr>
<td>HID=PNP0301</td>
<td></td>
</tr>
<tr>
<td></td>
<td>The UID parameter is an integer. It is optional for input but required for display. The default value is 0.</td>
</tr>
<tr>
<td>Type: 2 (ACPI Device Path)</td>
<td>Serial(UID)</td>
</tr>
<tr>
<td>SubType: 1 (ACPI Device Path)</td>
<td></td>
</tr>
<tr>
<td>HID=PNP0501</td>
<td></td>
</tr>
<tr>
<td></td>
<td>The UID parameter is an integer. It is optional for input but required for display. The default value is 0.</td>
</tr>
<tr>
<td>Type: 2 (ACPI Device Path)</td>
<td>ParallelPort(UID)</td>
</tr>
<tr>
<td>SubType: 1 (ACPI Device Path)</td>
<td></td>
</tr>
<tr>
<td>HID=PNP0401</td>
<td></td>
</tr>
<tr>
<td></td>
<td>The UID parameter is an integer. It is optional for input but required for display. The default value is 0.</td>
</tr>
<tr>
<td>Type: 2 (ACPI Device Path)</td>
<td>AcpiEx(HID,CID,UID,HIDSTR,CIDSTR,UIDSTR)</td>
</tr>
<tr>
<td>SubType: 2 (ACPI Expanded Device Path)</td>
<td></td>
</tr>
<tr>
<td></td>
<td>AcpiEx(HID,HIDSTR,CID,CIDSTR,UID,UIDSTR)[(Display Only)]</td>
</tr>
<tr>
<td></td>
<td>The HID parameter is an EISAID. The default value is 0. Either HID or HIDSTR must be present.</td>
</tr>
<tr>
<td></td>
<td>The CID parameter is an EISAID. The default value is 0. Either CID must be 0 or CIDSTR must be empty.</td>
</tr>
<tr>
<td></td>
<td>The UID parameter is an integer. The default value is 0. Either UID must be 0 or UIDSTR must be empty.</td>
</tr>
<tr>
<td></td>
<td>The HIDSTR is a string. The default value is the empty string. Either HID or HIDSTR must be present.</td>
</tr>
<tr>
<td></td>
<td>The CIDSTR is a string. The default value is an empty string. Either CID must be 0 or CIDSTR must be empty.</td>
</tr>
<tr>
<td></td>
<td>The UIDSTR is a string. The default value is an empty string. Either UID must be 0 or UIDSTR must be empty.</td>
</tr>
<tr>
<td>Type: 2 (ACPI Device Path)</td>
<td>AfpiExp(HID,CID,UIDSTR)</td>
</tr>
<tr>
<td>SubType: 2 (ACPI Expanded Device Path)</td>
<td></td>
</tr>
<tr>
<td>HIDSTR=empty</td>
<td></td>
</tr>
<tr>
<td>CIDSTR=empty</td>
<td></td>
</tr>
<tr>
<td>UID STR!=empty</td>
<td></td>
</tr>
<tr>
<td></td>
<td>The HID parameter is an EISAID. It is required.</td>
</tr>
<tr>
<td></td>
<td>The CID parameter is an EISAID. It is optional and has a default value of 0.</td>
</tr>
<tr>
<td></td>
<td>The UIDSTR parameter is a string. If UID is 0 and UIDSTR is empty, then use AcpiEx format.</td>
</tr>
<tr>
<td>Type: 2 (ACPI Device Path)</td>
<td>PciRoot(UID, UIDSTR) (Display Only)</td>
</tr>
<tr>
<td>SubType: 2 (ACPI Expanded Device Path)</td>
<td></td>
</tr>
<tr>
<td>HID=PNP0A03 or CID=PNP0A03 and HID != PNP0A08.</td>
<td></td>
</tr>
<tr>
<td></td>
<td>The DisplayDevice parameter is an Integer. There may be one or more, separated by a comma.</td>
</tr>
<tr>
<td>Device Node Type/SubType/Other</td>
<td>Description</td>
</tr>
<tr>
<td>-------------------------------</td>
<td>-------------</td>
</tr>
<tr>
<td>Type: 3 MessagingPath</td>
<td><code>Msg(subtype, data)</code></td>
</tr>
<tr>
<td>(when subtype is not recognized)</td>
<td>The <code>subtype</code> is an integer from 0-255. The <code>data</code> is a hex dump.</td>
</tr>
<tr>
<td>Type: 3 (Messaging Device Path) SubType: 1 (ATAPI)</td>
<td><code>Ata(Controller,Drive,LUN)</code></td>
</tr>
<tr>
<td></td>
<td><code>Ata(LUN)</code> (Display only)</td>
</tr>
<tr>
<td></td>
<td>The <code>Controller</code> is either an integer with a value of 0 or 1 or else the keyword <code>Primary</code> (0) or <code>Secondary</code> (1). It is required. The <code>Drive</code> is either an integer with the value of 0 or 1 or else the keyword <code>Master</code> (0) or <code>Slave</code> (1). It is required. The <code>LUN</code> is a 16-bit integer. It is required.</td>
</tr>
<tr>
<td>Type: 3 (Messaging Device Path) SubType: 2 (SCSI)</td>
<td><code>Scsi(PUN,LUN)</code></td>
</tr>
<tr>
<td></td>
<td>The <code>PUN</code> is an integer between 0 and 65535 and is required. The <code>LUN</code> is an integer between 0 and 65535 and is required.</td>
</tr>
<tr>
<td>Type: 3 (Messaging Device Path) SubType: 3 (Fibre Channel)</td>
<td><code>Fibre(WWN,LUN)</code></td>
</tr>
<tr>
<td></td>
<td>The <code>WWN</code> is a 64-bit unsigned integer and is required. The <code>LUN</code> is a 64-bit unsigned integer and is required.</td>
</tr>
<tr>
<td>Type: 3 (Messaging Device Path) SubType: 21 (Fibre Channel Ex)</td>
<td><code>FibreEx(WWN,LUN)</code></td>
</tr>
<tr>
<td></td>
<td>The <code>WWN</code> is an 8 byte array that is displayed in hexadecimal format with byte 0 first (i.e., on the left) and byte 7 last (i.e., on the right), and is required. The <code>LUN</code> is an 8 byte array that is displayed in hexadecimal format with byte 0 first (i.e., on the left) and byte 7 last (i.e., on the right), and is required.</td>
</tr>
<tr>
<td>Type: 3 (Messaging Device Path) SubType: 4 (1394)</td>
<td><code>I1394(GUID)</code></td>
</tr>
<tr>
<td></td>
<td>The <code>GUID</code> is a GUID and is required.</td>
</tr>
<tr>
<td>Type: 3 (Messaging Device Path) SubType: 5 (USB)</td>
<td><code>USB(Port,Interface)</code></td>
</tr>
<tr>
<td></td>
<td>The <code>Port</code> is an integer between 0 and 255 and is required. The <code>Interface</code> is an integer between 0 and 255 and is required.</td>
</tr>
<tr>
<td>Type: 3 (Messaging Device Path) SubType: 6 (I2O)</td>
<td><code>I2O(TID)</code></td>
</tr>
<tr>
<td></td>
<td>The <code>TID</code> is an integer and is required.</td>
</tr>
<tr>
<td>Type: 3 (Messaging Device Path) SubType: 9 (Infiniband)</td>
<td><code>Infiniband(Flags, GUID, ServiceId, TargetId, DeviceId)</code></td>
</tr>
</tbody>
</table>

  * `Flags` is an integer. `GUID` is a guid. `ServiceId`, `TargetId` and `DeviceId` are 64-bit unsigned integers. All fields are required. |
<p>| Type: 3 (Messaging Device Path) SubType: 10 (Vendor) | <code>VenMsg(Guid, Data)</code> |
|                               | The <code>Guid</code> is a GUID and is required. The <code>Data</code> is a Hex Dump and is option. The default value is zero bytes. |
| Type: 3 (Messaging Device Path) SubType: 10 (Vendor) GUID=EFI_PC_ANSI_GUID | <code>VenPcAnsi()</code> |</p>
<table>
<thead>
<tr>
<th>Device Node Type/SubType/Other</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Type: 3 (Messaging Device Path)</td>
<td>VenVt100()</td>
</tr>
<tr>
<td>SubType: 10 (Vendor)</td>
<td>GUID=EFI_VT_100_GIUD</td>
</tr>
<tr>
<td>Type: 3 (Messaging Device Path)</td>
<td>VenVt100Plus()</td>
</tr>
<tr>
<td>SubType: 10 (Vendor)</td>
<td>GUID=EFI_VT_100_PLUS_GIUD</td>
</tr>
<tr>
<td>Type: 3 (Messaging Device Path)</td>
<td>VenUtf8()</td>
</tr>
<tr>
<td>SubType: 10 (Vendor)</td>
<td>GUID=EFI_VT_UTF8_GUID</td>
</tr>
<tr>
<td>Type: 3 (Messaging Device Path)</td>
<td>UartFlowCtrl(Value)</td>
</tr>
<tr>
<td>SubType: 10 (Vendor)</td>
<td>GUID=DEVICE_PATH_MESSAGING_UART_FLOW_CONTROL</td>
</tr>
<tr>
<td>Type: 3 (Messaging Device Path)</td>
<td>SAS(Address, LUN, RTP, SASSATA, Location, Connect, DriveBay, Reserved)</td>
</tr>
<tr>
<td>SubType: 10 (Serial Attached SCSI)</td>
<td>Vendor GUID: d487ddb4-008b-11d9-afdc-001083ffca4d</td>
</tr>
<tr>
<td>Type: 3 (Messaging Device Path)</td>
<td>DebugPort()</td>
</tr>
<tr>
<td>SubType: 10 (Vendor)</td>
<td>GUID=EFI_DEBUGPORT_PROTOCOL_GUID</td>
</tr>
<tr>
<td>Type: 3 (Messaging Device Path)</td>
<td>MAC(MacAddr, IfType)</td>
</tr>
<tr>
<td>SubType: 11 (MAC Address)</td>
<td></td>
</tr>
</tbody>
</table>

- The Value is either an integer with the value 0, 1 or 2 or the keywords XonXoff (2) or Hardware (1) or None (0).
- The Address is a 64-bit unsigned integer representing the SAS Address and is required.
- The LUN is a 64-bit unsigned integer representing the Logical Unit Number and is optional. The default value is 0.
- The RTP is a 16-bit unsigned integer representing the Relative Target Port and is optional. The default value is 0.
- The SASSATA is a keyword SAS or SATA or NoTopology or an unsigned 16-bit integer and is optional. The default is NoTopology. If NoTopology or an integer are specified, then Location, Connect and DriveBay are prohibited. If SAS or SATA is specified, then Location and Connect are required, but DriveBay is optional. If an integer is specified, then the topology information is filled with the integer value.
- The Location is an integer between 0 and 1 or else the keyword Internal (0) or External (1) and is optional. If SASSATA is an integer or NoTopology, it is prohibited. The default value is 0.
- The Connect is an integer between 0 and 3 or else the keyword Direct (0) or Expanded (1) and is optional. If SASSATA is an integer or NoTopology, it is prohibited. The default value is 0.
- The DriveBay is an integer between 1 and 256 and is optional unless SASSATA is an integer or NoTopology, in which case it is prohibited. The Reserved field is an integer and is optional. The default value is 0.
- The MacAddr is a Hex Dump and is required. If IfType is 0 or 1, then the MacAddr must be exactly six bytes.
- The IfType is an integer from 0-255 and is optional. The default is zero.
<table>
<thead>
<tr>
<th>Device Node Type/SubType/Other</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Type: 3 (Messaging Device Path) SubType: 12 (IPv4)</td>
<td>IPv4(RemoteIp, Protocol, Type, LocalIp, GatewayIPAddress, SubnetMask) IPv4(RemoteIp) (Display Only)</td>
</tr>
<tr>
<td></td>
<td>The RemoteIp is an IP Address and is required. The Protocol is an integer between 0 and 255 or else the keyword UDP (17) or TCP (6). The default value is UDP. The Type is a keyword, either Static (1) or DHCP (0). It is optional. The default value is DHCP. The LocalIp is an IP Address and is optional. The default value is all zeroes. The GatewayIPAddress is an IP Address and is optional. The default value is all zeroes. The SubnetMask is an IP Address and is optional. The default value is all zeroes.</td>
</tr>
<tr>
<td>Type: 3 (Messaging Device Path) SubType: 13 (IPv6)</td>
<td>IPv6(RemoteIp, Protocol, IPAddressOrigin, LocalIp, GatewayIPAddress, SubnetMask) IPv6(RemoteIp) (Display Only)</td>
</tr>
<tr>
<td></td>
<td>The RemoteIp is an IPv6 Address and is required. The Protocol is an integer between 0 and 255 or else the keyword UDP (17) or TCP (6). The default value is UDP. The IPAddressOrigin is a keyword, could be Static (0), StatelessAutoConfigure (1), or StatefulAutoConfigure (2). The LocalIp is the IPv6 Address and is optional. The default value is all zeroes. The GatewayIPAddress is an IPv6 Address. The PrefixLength is the prefix length of the Local IPv6 Address. The GatewayIPAddress is the IPv6 Address of the Gateway.</td>
</tr>
<tr>
<td>Type: 3 (Messaging Device Path) SubType: 14 (UART)</td>
<td>Uart(Baud, DataBits, Parity, StopBits)</td>
</tr>
<tr>
<td></td>
<td>The Baud is a 64-bit integer and is optional. The default value is 115200. The DataBits is an integer from 0 to 255 and is optional. The default value is 8. The Parity is either an integer from 0-255 or else a keyword and should be D (0), N (1), E (2), O (3), M (4) or S (5). It is optional. The default value is 0. The StopBits is a either an integer from 0-255 or else a keyword and should be D (0), 1 (1), 1.5 (2), 2 (3). It is optional. The default value is 0.</td>
</tr>
<tr>
<td>Type: 3 (Messaging Device Path) SubType: 15 (USB Class)</td>
<td>UsbClass(VID, PID, Class, SubClass, Protocol)</td>
</tr>
<tr>
<td></td>
<td>The VID is an integer between 0 and 65535 and is optional. The default value is 0xFFFF. The PID is an integer between 0 and 65535 and is optional. The default value is 0xFFFF. The Class is an integer between 0 and 255 and is optional. The default value is 0xFF. The SubClass is an integer between 0 and 255 and is optional. The default value is 0xFF. The Protocol is an integer between 0 and 255 and is optional. The default value is 0xFF.</td>
</tr>
<tr>
<td>Device Node Type/SubType/Other</td>
<td>Description</td>
</tr>
<tr>
<td>-------------------------------</td>
<td>-------------</td>
</tr>
</tbody>
</table>
| Type: 3 (Messaging Device Path) SubType: 15 (USB Class) Class 1 | **UsbAudio(VID,PID,SubClass,Protocol)**  
The VID is an integer between 0 and 65535 and is optional. The default value is 0xFFFF.  
The PID is an integer between 0 and 65535 and is optional. The default value is 0xFFFF.  
The SubClass is an integer between 0 and 255 and is optional. The default value is 0xFF.  
The Protocol is an integer between 0 and 255 and is optional. The default value is 0xFF. |
| Type: 3 (Messaging Device Path) SubType: 15 (USB Class) Class 2 | **UsbCDCControl(VID,PID,SubClass,Protocol)**  
The VID is an optional integer between 0 and 65535 and is optional. The default value is 0xFFFF.  
The PID is an optional integer between 0 and 65535 and is optional. The default value is 0xFFFF.  
The SubClass is an optional integer between 0 and 255 and is optional. The default value is 0xFF.  
The Protocol is an optional integer between 0 and 255 and is optional. The default value is 0xFF. |
| Type: 3 (Messaging Device Path) SubType: 15 (USB Class) Class 3 | **UsbHID(VID,PID,SubClass,Protocol)**  
The VID is an integer between 0 and 65535 and is optional. The default value is 0xFFFF.  
The PID is an integer between 0 and 65535 and is optional. The default value is 0xFFFF.  
The SubClass is an integer between 0 and 255 and is optional. The default value is 0xFF.  
The Protocol is an integer between 0 and 255 and is optional. The default value is 0xFF. |
| Type: 3 (Messaging Device Path) SubType: 15 (USB Class) Class 6 | **UsbImage(VID,PID,SubClass,Protocol)**  
The VID is an integer between 0 and 65535 and is optional. The default value is 0xFFFF.  
The PID is an integer between 0 and 65535 and is optional. The default value is 0xFFFF.  
The SubClass is an integer between 0 and 255 and is optional. The default value is 0xFF.  
The Protocol is an integer between 0 and 255 and is optional. The default value is 0xFF. |
| Type: 3 (Messaging Device Path) SubType: 15 (USB Class) Class 7 | **UsbPrinter(VID,PID,SubClass,Protocol)**  
The VID is an integer between 0 and 65535 and is optional. The default value is 0xFFFF.  
The PID is an integer between 0 and 65535 and is optional. The default value is 0xFFFF.  
The SubClass is an integer between 0 and 255 and is optional. The default value is 0xFF.  
The Protocol is an integer between 0 and 255 and is optional. The default value is 0xFF. |
<table>
<thead>
<tr>
<th>Device Node Type/SubType/Other</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Type: 3 (Messaging Device Path) SubType: 15 (USB Class) Class 8</td>
<td><strong>UsbMassStorage(VID, PID, SubClass, Protocol)</strong> &lt;br&gt;The VID is an integer between 0 and 65535 and is optional. The default value is 0xFFFF. &lt;br&gt;The PID is an integer between 0 and 65535 and is optional. The default value is 0xFFFF. &lt;br&gt;The SubClass is an integer between 0 and 255 and is optional. The default value is 0xFF. &lt;br&gt;The Protocol is an integer between 0 and 255 and is optional. The default value is 0xFF.</td>
</tr>
<tr>
<td>Type: 3 (Messaging Device Path) SubType: 15 (USB Class) Class 9</td>
<td><strong>UsbHub(VID, PID, SubClass, Protocol)</strong> &lt;br&gt;The VID is an integer between 0 and 65535 and is optional. The default value is 0xFFFF. &lt;br&gt;The PID is an integer between 0 and 65535 and is optional. The default value is 0xFFFF. &lt;br&gt;The SubClass is an integer between 0 and 255 and is optional. The default value is 0xFF. &lt;br&gt;The Protocol is an integer between 0 and 255 and is optional. The default value is 0xFF.</td>
</tr>
<tr>
<td>Type: 3 (Messaging Device Path) SubType: 15 (USB Class) Class 10</td>
<td><strong>UsbCDCData(VID, PID, SubClass, Protocol)</strong> &lt;br&gt;The VID is an integer between 0 and 65535 and is optional. The default value is 0xFFFF. &lt;br&gt;The PID is an integer between 0 and 65535 and is optional. The default value is 0xFFFF. &lt;br&gt;The SubClass is an integer between 0 and 255 and is optional. The default value is 0xFF. &lt;br&gt;The Protocol is an integer between 0 and 255 and is optional. The default value is 0xFF.</td>
</tr>
<tr>
<td>Type: 3 (Messaging Device Path) SubType: 15 (USB Class) Class 11</td>
<td><strong>UsbSmartCard(VID, PID, SubClass, Protocol)</strong> &lt;br&gt;The VID is an integer between 0 and 65535 and is optional. The default value is 0xFFFF. &lt;br&gt;The PID is an integer between 0 and 65535 and is optional. The default value is 0xFFFF. &lt;br&gt;The SubClass is an integer between 0 and 255 and is optional. The default value is 0xFF. &lt;br&gt;The Protocol is an integer between 0 and 255 and is optional. The default value is 0xFF.</td>
</tr>
<tr>
<td>Type: 3 (Messaging Device Path) SubType: 15 (USB Class) Class 14</td>
<td><strong>UsbVideo(VID, PID, SubClass, Protocol)</strong> &lt;br&gt;The VID is an integer between 0 and 65535 and is optional. The default value is 0xFFFF. &lt;br&gt;The PID is an integer between 0 and 65535 and is optional. The default value is 0xFFFF. &lt;br&gt;The SubClass is an integer between 0 and 255 and is optional. The default value is 0xFF. &lt;br&gt;The Protocol is an integer between 0 and 255 and is optional. The default value is 0xFF.</td>
</tr>
<tr>
<td>Device Node Type/SubType/Other</td>
<td>Description</td>
</tr>
<tr>
<td>--------------------------------</td>
<td>-------------</td>
</tr>
</tbody>
</table>
| **Type: 3 (Messaging Device Path)**  
SubType: 15 (USB Class)  
Class 220 | **UsbDiagnostic(VID,PID,SubClass,Protocol)**  
The VID is an integer between 0 and 65535 and is optional. The default value is 0xFFFF.  
The PID is an integer between 0 and 65535 and is optional. The default value is 0xFFFF.  
The SubClass is an integer between 0 and 255 and is optional. The default value is 0xFF.  
The Protocol is an integer between 0 and 255 and is optional. The default value is 0xFF. |
| **Type: 3 (Messaging Device Path)**  
SubType: 15 (USB Class)  
Class 224 | **UsbWireless(VID,PID,SubClass,Protocol)**  
The VID is an integer between 0 and 65535 and is optional. The default value is 0xFFFF.  
The PID is an integer between 0 and 65535 and is optional. The default value is 0xFFFF.  
The SubClass is an integer between 0 and 255 and is optional. The default value is 0xFF.  
The Protocol is an integer between 0 and 255 and is optional. The default value is 0xFF. |
| **Type: 3 (Messaging Device Path)**  
SubType: 15 (USB Class)  
Class 254  
SubClass: 1 | **UsbDeviceFirmwareUpdate(VID,PID,Protocol)**  
The VID is an integer between 0 and 65535 and is optional. The default value is 0xFFFF.  
The PID is an integer between 0 and 65535 and is optional. The default value is 0xFFFF.  
The Protocol is an integer between 0 and 255 and is optional. The default value is 0xFF. |
| **Type: 3 (Messaging Device Path)**  
SubType: 15 (USB Class)  
Class 254  
SubClass: 2 | **UsbIrdaBridge(VID,PID,Protocol)**  
The VID is an integer between 0 and 65535 and is optional. The default value is 0xFFFF.  
The PID is an integer between 0 and 65535 and is optional. The default value is 0xFFFF.  
The Protocol is an integer between 0 and 255 and is optional. The default value is 0xFF. |
| **Type: 3 (Messaging Device Path)**  
SubType: 15 (USB Class)  
Class 254  
SubClass: 3 | **UsbTestAndMeasurement(VID,PID,Protocol)**  
The VID is an integer between 0 and 65535 and is optional. The default value is 0xFFFF.  
The PID is an integer between 0 and 65535 and is optional. The default value is 0xFFFF.  
The Protocol is an integer between 0 and 255 and is optional. The default value is 0xFF. |
| **Type: 3 (Messaging Device Path)**  
SubType: 16 (USB WWID Class) | **UsbWwid(VID,PID,InterfaceNumber,"WWID")**  
The VID is an integer between 0 and 65535 and is required.  
The PID is an integer between 0 and 65535 and is required.  
The InterfaceNumber is an integer between 0 and 255 and is required.  
The WWID is a string and is required. |
<table>
<thead>
<tr>
<th>Device Node Type/SubType/Other</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Type: 3 (Messaging Device Path) SubType: 17 (Logical Unit Class)</td>
<td><strong>Unit(LUN)</strong>&lt;br&gt;The LUN is an integer and is required.</td>
</tr>
<tr>
<td>Type: 3 (Messaging Device Path) SubType: 18 (SATA)</td>
<td><strong>Sata</strong>(HPN, PMPN, LUN)&lt;br&gt;The HPN is an integer between 0 and 65534 and is required. The PMPN is an integer between 0 and 65535 and is optional. If not provided, the default is 0xFFFF, which implies no port multiplier. The LUN is a 16-bit integer. It is required. Note that LUN is applicable to ATAPI devices only, and most ATAPI devices assume LUN=0</td>
</tr>
<tr>
<td>Type: 3 (Messaging Device Path) SubType: 19 (iSCSI)</td>
<td><strong>iSCSI</strong>(TargetName, PortalGroup, LUN, HeaderDigest, DataDigest, Authentication, Protocol)&lt;br&gt;The TargetName is a string and is required. The PortalGroup is an unsigned 16-bit integer and is required. The LUN is an 8 byte array that is displayed in hexadecimal format with byte 0 first (i.e., on the left) and byte 7 last (i.e., on the right), and is required. The HeaderDigest is a keyword None or CRC32C is optional. The default is None. The DataDigest is a keyword None or CRC32C is optional. The default is None. The Authentication is a keyword None or CHAP_BI or CHAP_UNI and optional. The default is None. The Protocol defines the network protocol used by iSCSI and is optional. The default is TCP.</td>
</tr>
<tr>
<td>Type: 3 (Messaging Device Path) SubType: 20 (VLAN)</td>
<td><strong>Vlan</strong>(VlanId)</td>
</tr>
<tr>
<td>Type: 3 (Messaging Device Path) SubType: 22 (Serial Attached SCSI Ex)</td>
<td><strong>SasEx</strong>(Address, LUN, RTP, SASSATA, Location, Connect, DriveBay)  &lt;br&gt;The Address is an 8 byte array that is displayed in hexadecimal format with byte 0 first (i.e., on the left) and byte 7 last (i.e., on the right), and is required. The LUN is an 8 byte array that is displayed in hexadecimal format with byte 0 first (i.e., on the left) and byte 7 last (i.e., on the right), and is optional. The default value is 0. The RTP is a 16-bit unsigned integer representing the Relative Target Port and is optional. The default value is 0. The SASSATA is a keyword SAS or SATA or NoTopology or an unsigned 16-bit integer and is optional. The default is NoTopology. If NoTopology or an integer are specified, then Location, Connect and DriveBay are prohibited. If SAS or SATA is specified, then Location and Connect are required, but DriveBay is optional. If an integer is specified, then the topology information is filled with the integer value. The Location is an integer between 0 and 1 or else the keyword Internal (0) or External (1) and is optional. If SASSATA is an integer or NoTopology, it is prohibited. The default value is 0. The Connect is an integer between 0 and 3 or else the keyword Direct (0) or Expanded (1) and is optional. If SASSATA is an integer or NoTopology, it is prohibited. The default value is 0. The DriveBay is an integer between 1 and 256 and is optional unless SASSATA is an integer or NoTopology, in which case it is prohibited.</td>
</tr>
<tr>
<td>Device Node Type/SubType/Other</td>
<td>Description</td>
</tr>
<tr>
<td>-------------------------------------------------------------------</td>
<td>-------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------</td>
</tr>
<tr>
<td>Type: 3 (Messaging Device Path)</td>
<td><strong>NVMe(NSID,EUI)</strong></td>
</tr>
<tr>
<td>SubType: 23 (NVMe Express Namespace)</td>
<td>The NSID is a namespace identifier that is displayed in hexadecimal format with an integer value between 0 and 0xFFFFFFFF. The EUI is the IEEE Extended Unique Identifier (EUI-64) that is displayed in hexadecimal format represented as a set of octets separated by dashes (hexadecimal notation), e.g., FF-FF-FF-FF-FF-FF-FF.</td>
</tr>
<tr>
<td>Type: 3 (Messaging Device Path)</td>
<td><strong>Uri(Uri)</strong></td>
</tr>
<tr>
<td>SubType: 24 (URI)</td>
<td>The Uri is optional.</td>
</tr>
<tr>
<td>Type: 3 (Messaging Device Path)</td>
<td><strong>UFS(PUN,LUN)</strong></td>
</tr>
<tr>
<td>SubType: 25 (Universal Flash Storage)</td>
<td>The PUN is 0 for current UFS2.0 spec. For future UFS specs which support multiple devices on a UFS port, it would reflect the device ID on the UFS port. The LUN is 0-7 for common LUNs or 81h, D0h, B0h and C4h for well-known LUNs supported by UFS.</td>
</tr>
<tr>
<td>Type: 3 (Messaging Device Path)</td>
<td><strong>SD(Slot Number)</strong></td>
</tr>
<tr>
<td>SubType: 26 (SD)</td>
<td>SlotNumber is an integer. It is optional and has a default value of 0.</td>
</tr>
<tr>
<td>Type: 3 (Messaging Device Path)</td>
<td><strong>Bluetooth(BD_ADDR)</strong></td>
</tr>
<tr>
<td>SubType: 27 (Bluetooth)</td>
<td>BD_ADDR is HEX dump of 48-bit Bluetooth device address.</td>
</tr>
<tr>
<td>Type: 3 (Messaging Device Path)</td>
<td><strong>Wi-Fi(SSID)</strong></td>
</tr>
<tr>
<td>SubType: 28 (Wi-Fi)</td>
<td>The SSID is a string and is required.</td>
</tr>
<tr>
<td>Type: 3 (Messaging Device Path)</td>
<td><strong>eMMC(SlotNumber)</strong></td>
</tr>
<tr>
<td>SubType: 29 (eMMC)</td>
<td>SlotNumber is an integer. It is optional and has a default value of 0.</td>
</tr>
<tr>
<td>Type: 3 (Messaging Device Path)</td>
<td><strong>BluetoothLE(BD_ADDR, AddressType)</strong></td>
</tr>
<tr>
<td>SubType: 30 (BluetoothLE)</td>
<td>BD_ADDR is HEX dump of 48-bit Bluetooth device address. AddressType is an integer.</td>
</tr>
<tr>
<td>Type: 3 (Messaging Device Path)</td>
<td><strong>Dns(DnsServerIp[, DnsServerIp…])</strong></td>
</tr>
<tr>
<td>SubType: 31 (DNS)</td>
<td>DnsServerIp is optional. It is the IP address of DNS server.</td>
</tr>
<tr>
<td>Type: 3 (Messaging Device Path)</td>
<td><strong>RestService(RestExServiceType, AccessMode)</strong></td>
</tr>
<tr>
<td>SubType: 32 (REST Service)</td>
<td>For vendor-specific REST service: RestService(RestExServiceType, AccessMode, VendorRestServiceGuid, VendorDefinedData) RestExServiceType is 0xff in this case.</td>
</tr>
<tr>
<td>Type: 4 (Media Device Path)</td>
<td><strong>MediaPath(subtype, data)</strong></td>
</tr>
<tr>
<td>(when subtype is not recognized)</td>
<td>The subtype is an integer from 0-255 and is required. The data is a hex dump.</td>
</tr>
<tr>
<td>Device Node Type/SubType/Other</td>
<td>Description</td>
</tr>
<tr>
<td>-------------------------------</td>
<td>-------------</td>
</tr>
<tr>
<td>Type: 4 (Media Device Path)</td>
<td></td>
</tr>
<tr>
<td>SubType: 1 (Hard Drive)</td>
<td>HD(Partition,Type,Sigature,Start, Size)</td>
</tr>
<tr>
<td></td>
<td>HD(Partition,Type,Sigature) (Display Only)</td>
</tr>
<tr>
<td></td>
<td>The Partition is an integer representing the partition number. It is optional and the default is 0. If Partition is 0, then Start and Size are prohibited.</td>
</tr>
<tr>
<td></td>
<td>The Type is an integer between 0-255 or else the keyword MBR (1) or GPT (2). The type is optional and the default is 2.</td>
</tr>
<tr>
<td></td>
<td>The Signature is an integer if Type is 1 or else GUID if Type is 2. The signature is required.</td>
</tr>
<tr>
<td></td>
<td>The Start is a 64-bit unsigned integer. It is prohibited if Partition is 0. Otherwise it is required.</td>
</tr>
<tr>
<td></td>
<td>The Size is a 64-bit unsigned integer. It is prohibited if Partition is 0. Otherwise it is required.</td>
</tr>
<tr>
<td>Type: 4 (Media Device Path)</td>
<td>CDROM(Entry,Start,Size)</td>
</tr>
<tr>
<td>SubType: 2 (CD-ROM)</td>
<td>CDROM(Entry) (Display Only)</td>
</tr>
<tr>
<td></td>
<td>The Entry is an integer representing the Boot Entry from the Boot Catalog. It is optional and the default is 0.</td>
</tr>
<tr>
<td></td>
<td>The Start is a 64-bit integer and is required.</td>
</tr>
<tr>
<td></td>
<td>The Size is a 64-bit integer and is required.</td>
</tr>
<tr>
<td>Type: 4 (Media Device Path)</td>
<td>VenMedia(GUID, Data)</td>
</tr>
<tr>
<td>SubType: 3 (Vendor)</td>
<td>The Guid is a GUID and is required.</td>
</tr>
<tr>
<td></td>
<td>The Data is a Hex Dump and is optional. The default value is zero bytes.</td>
</tr>
<tr>
<td>Type: 4 (Media Device Path)</td>
<td>String</td>
</tr>
<tr>
<td>SubType: 4 (File Path)</td>
<td>The String is the file path and is a string.</td>
</tr>
<tr>
<td>Type: 4 (Media Device Path)</td>
<td>Media(Guid)</td>
</tr>
<tr>
<td>SubType: 5 (Media Protocol)</td>
<td>The Guid is a GUID and is required.</td>
</tr>
<tr>
<td>Type: 4 (Media Device Path)</td>
<td>Contents are defined in the UEFI PI Specification.</td>
</tr>
<tr>
<td>SubType: 6 (PIWG Firmware File)</td>
<td></td>
</tr>
<tr>
<td>Type: 4 (Media Device Path)</td>
<td>Contents are defined in the UEFI PI Specification.</td>
</tr>
<tr>
<td>SubType: 7 (PIWG Firmware Volume)</td>
<td></td>
</tr>
<tr>
<td>Type: 4 (Media Device Path)</td>
<td>Offset(StartingOffset,EndingOffset)</td>
</tr>
<tr>
<td>SubType: 8 (Relative Offset Range)</td>
<td>The StartingOffset is an unsigned 64-bit integer. The EndingOffset is an unsigned 64-bit integer.</td>
</tr>
<tr>
<td>Type: 4 (Media Device Path)</td>
<td>RamDisk (StartingAddress,EndingAddress,DiskInstance,DiskTypeGuid)</td>
</tr>
<tr>
<td>SubType: 9 (RAM Disk)</td>
<td>The StartingAddress and EndingAddress are both 64-bit integers and are both required.</td>
</tr>
<tr>
<td></td>
<td>The DiskInstance is a 16-bit integer and is optional. The default value is 0.</td>
</tr>
<tr>
<td></td>
<td>The DiskTypeGuid is a GUID and is required.</td>
</tr>
<tr>
<td>Type: 4 (Media Device Path)</td>
<td>VirtualDisk StartingAddress,EndingAddress,DiskInstance)</td>
</tr>
<tr>
<td>SubType: 9 (RAM Disk)</td>
<td>The StartingAddress and EndingAddress are both 64-bit integers and are both required.</td>
</tr>
<tr>
<td></td>
<td>The DiskInstance is a 16-bit integer and is optional. The default value is 0.</td>
</tr>
</tbody>
</table>
10.6.2 Device Path to Text Protocol

**EFI_DEVICE_PATH_TO_TEXT_PROTOCOL**

**Summary**

Convert device nodes and paths to text.

**GUID**

```
#define EFI_DEVICE_PATH_TO_TEXT_PROTOCOL_GUID {
  0x8b843e20, 0x8132, 0x4852,
  0x90, 0xcc, 0x55, 0x1a, 0x4e, 0x4a, 0x7f, 0x1c}
```

**Protocol Interface Structure**

```
typedef struct _EFI_DEVICE_PATH_TO_TEXT_PROTOCOL {
  EFI_DEVICE_PATH_TO_TEXT_NODE ConvertDeviceNodeToText;
  EFI_DEVICE_PATH_TO_TEXT_PATH ConvertDevicePathToText;
} EFI_DEVICE_PATH_TO_TEXT_PROTOCOL;
```

**Parameters**

- `ConvertDeviceNodeToText` Converts a device node to text.
- `ConvertDevicePathToText` Converts a device path to text.

---

<table>
<thead>
<tr>
<th>Device Node Type/SubType/Other</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Type: 4 (Media Device Path)</td>
<td>VirtualCD(StartingAddress,EndingAddress,DiskInstance)</td>
</tr>
<tr>
<td>SubType: 9 (RAM Disk)</td>
<td>The <code>StartingAddress</code> and <code>EndingAddress</code> are both 64-bit integers and are both required. The <code>DiskInstance</code> is a 16-bit integer and is optional. The default value is 0.</td>
</tr>
<tr>
<td>Disk Type GUID=</td>
<td>EFI_VIRTUAL_CD_GUID</td>
</tr>
<tr>
<td>Type: 4 (Media Device Path)</td>
<td>PersistentVirtualDisk (StartingAddress,EndingAddress,DiskInstance)</td>
</tr>
<tr>
<td>SubType: 9 (RAM Disk)</td>
<td>The <code>StartingAddress</code> and <code>EndingAddress</code> are both 64-bit integers and are both required. The <code>DiskInstance</code> is a 16-bit integer and is optional. The default value is 0.</td>
</tr>
<tr>
<td>Disk Type GUID=</td>
<td>EFI_PERSISTENT_VIRTUAL_DISK_GUID</td>
</tr>
<tr>
<td>Type: 4 (Media Device Path)</td>
<td>PersistentVirtualCD (StartingAddress,EndingAddress,DiskInstance)</td>
</tr>
<tr>
<td>SubType: 9 (RAM Disk)</td>
<td>The <code>StartingAddress</code> and <code>EndingAddress</code> are both 64-bit integers and are both required. The <code>DiskInstance</code> is a 16-bit integer and is optional. The default value is 0.</td>
</tr>
<tr>
<td>Disk Type GUID=</td>
<td>EFI_PERSISTENT_VIRTUAL_CD_GUID</td>
</tr>
<tr>
<td>Type: 5 (Media Device Path)</td>
<td>BbsPath(subtype, data)</td>
</tr>
<tr>
<td>(when subtype is not recognized)</td>
<td>The <code>subtype</code> is an integer from 0-255. The <code>data</code> is a hex dump.</td>
</tr>
<tr>
<td>Type: 5 (BIOS Boot Specification Device Path)</td>
<td>BBS(Type, Id.Flags)</td>
</tr>
<tr>
<td>SubType: 1 (BBS 1.01)</td>
<td>BBS(Type, Id) (Display Only)</td>
</tr>
<tr>
<td></td>
<td>The <code>Type</code> is an integer from 0-65535 or else one of the following keywords: Floppy (1), HD (2), CDROM (3), PCMCIA (4), USB (5), Network (6). It is required. The <code>Id</code> is a string and is required. The <code>Flags</code> are an integer and are optional. The default value is 0.</td>
</tr>
</tbody>
</table>
Description
The **EFI_DEVICE_PATH_TO_TEXT_PROTOCOL** provides common utility functions for converting device nodes and device paths to a text representation.

**EFI_DEVICE_PATH_TO_TEXT_PROTOCOL.ConvertDeviceNodeToText()**

**Summary**
Convert a device node to its text representation.

**Prototype**
```c
typedef CHAR16* (EFIAPI *EFI_DEVICE_PATH_TO_TEXT_NODE) (  
    IN CONST EFI_DEVICE_PATH_PROTOCOL* DeviceNode,  
    IN BOOLEAN DisplayOnly,  
    IN BOOLEAN AllowShortcuts
);
```

**Parameters**
- **DeviceNode**: Points to the device node to be converted.
- **DisplayOnly**: If `DisplayOnly` is `TRUE`, then the shorter text representation of the display node is used, where applicable. If `DisplayOnly` is `FALSE`, then the longer text representation of the display node is used.
- **AllowShortcuts**: If `AllowShortcuts` is `TRUE`, then the shortcut forms of text representation for a device node can be used, where applicable.

**Description**
The `ConvertDeviceNodeToText` function converts a device node to its text representation and copies it into a newly allocated buffer.

The `DisplayOnly` parameter controls whether the longer (parseable) or shorter (display-only) form of the conversion is used.

The `AllowShortcuts` is `FALSE`, then the shortcut forms of text representation for a device node cannot be used. A shortcut form is one which uses information other than the type or subtype.

The memory is allocated from EFI boot services memory. It is the responsibility of the caller to free the memory allocated.

**Related Definitions**
**EFI_DEVICE_PATH_PROTOCOL** is defined in Section 10.2.

**Returns**
This function returns the pointer to the allocated text representation of the device node data or else NULL if `DeviceNode` was NULL or there was insufficient memory.
**EFI_DEVICE_PATH_TO_TEXT_PROTOCOL.ConvertDevicePathToText()**

**Summary**
Convert a device path to its text representation.

**Prototype**
```c
typedef CHAR16* (EFIAPI *EFI_DEVICE_PATH_TO_TEXT_PATH) (
    IN CONST EFI_DEVICE_PATH_PROTOCOL *DevicePath,
    IN BOOLEAN DisplayOnly,
    IN BOOLEAN AllowShortcuts
);
```

**Parameters**
- **DeviceNode** Points to the device path to be converted.
- **DisplayOnly** If `DisplayOnly` is TRUE, then the shorter text representation of the display node is used, where applicable. If `DisplayOnly` is FALSE, then the longer text representation of the display node is used.
- **AllowShortcuts** The `AllowShortcuts` is FALSE, then the shortcut forms of text representation for a device node cannot be used.

**Description**
This function converts a device path into its text representation and copies it into an allocated buffer.

The `DisplayOnly` parameter controls whether the longer (parseable) or shorter (display-only) form of the conversion is used.

The `AllowShortcuts` is `FALSE`, then the shortcut forms of text representation for a device node cannot be used. A shortcut form is one which uses information other than the type or subtype.

The memory is allocated from EFI boot services memory. It is the responsibility of the caller to free the memory allocated.

**Related Definitions**

*EFI_DEVICE_PATH_PROTOCOL* is defined in Section 10.2.

**Returns**
This function returns a pointer to the allocated text representation of the device node or NULL if `DevicePath` was NULL or there was insufficient memory.

**10.6.3 Device Path from Text Protocol**

**EFI_DEVICE_PATH_FROM_TEXT_PROTOCOL**

**Summary**
Convert text to device paths and device nodes.
GUID
#define EFI_DEVICE_PATH_FROM_TEXT_PROTOCOL_GUID  
{0x5c99a21,0xc70f,0x4ad2,\  
{0x8a,0x5f,0x35,0xdf,0x33,0x43,0xf5, 0x1e}}

Protocol Interface Structure
typedef struct _EFI_DEVICE_PATH_FROM_TEXT_PROTOCOL {
  EFI_DEVICE_PATH_FROM_TEXT_NODE ConvertTextToDeviceNode;
  EFI_DEVICE_PATH_FROM_TEXT_PATH ConvertTextToDevicePath;
} EFI_DEVICE_PATH_FROM_TEXT_PROTOCOL;

Parameters
  ConvertTextToDeviceNode Converts text to a device node.
  ConvertTextToDevicePath Converts text to a device path.

Description
The EFI_DEVICE_PATH_FROM_TEXT_PROTOCOL provides common utilities for converting text to device paths and device nodes.

EFI_DEVICE_PATH_FROM_TEXT_PROTOCOL.ConvertTextToDeviceNode()

Summary
Convert text to the binary representation of a device node.

Prototype
typedef EFI_DEVICE_PATH_PROTOCOL* (EFIAPI *EFI_DEVICE_PATH_FROM_TEXT_NODE) ( 
  IN CONST CHAR16* TextDeviceNode 
);

Parameters
  TextDeviceNode TextDeviceNode points to the text representation of a device node. Conversion starts with the first character and continues until the first non-device node character.

Description
This function converts text to its binary device node representation and copies it into an allocated buffer. The memory is allocated from EFI boot services memory. It is the responsibility of the caller to free the memory allocated.

Related Definitions
EFI_DEVICE_PATH_PROTOCOL is defined in Section 10.2.
Returns
This function returns a pointer to the EFI device node or NULL if TextDeviceNode is NULL or there was insufficient memory.

EFI_DEVICE_PATH_FROM_TEXT_PROTOCOL.ConvertTextToDevicePath()

Summary
Convert a text to its binary device path representation.

Prototype

typedef

    EFI_DEVICE_PATH_PROTOCOL*

    (EFIAPI *EFI_DEVICE_PATH_FROM_TEXT_PATH) (  
        IN CONST CHAR16* TextDevicePath
    );

Parameters

    TextDevicePath points to the text representation of a device path. Conversion starts with the first character and continues until the first non-device path character.

Description
This function converts text to its binary device path representation and copies it into an allocated buffer. The memory is allocated from EFI boot services memory. It is the responsibility of the caller to free the memory allocated.

Related Definitions
EFI_DEVICE_PATH_PROTOCOL is defined in Section 10.2.

Returns
This function returns a pointer to the allocated device path or NULL if TextDevicePath is NULL or there was insufficient memory.
11 - Protocols — UEFI Driver Model

EFI drivers that follow the UEFI Driver Model are not allowed to search for controllers to manage. When a specific controller is needed, the EFI boot service EFI_BOOT_SERVICES.ConnectController() is used along with the EFI_DRIVER_BINDING_PROTOCOL services to identify the best drivers for a controller. Once ConnectController() has identified the best drivers for a controller, the start service in the EFI_DRIVER_BINDING_PROTOCOL is used by ConnectController() to start each driver on the controller. Once a controller is no longer needed, it can be released with the EFI boot service EFI_BOOT_SERVICES.DisconnectController(). DisconnectController() calls the stop service in each EFI_DRIVER_BINDING_PROTOCOL to stop the controller.

The driver initialization routine of an UEFI driver is not allowed to touch any device hardware. Instead, it just installs an instance of the EFI_DRIVER_BINDING_PROTOCOL on the ImageHandle of the UEFI driver. The test to determine if a driver supports a given controller must be performed in as little time as possible without causing any side effects on any of the controllers it is testing. As a result, most of the controller initialization code is present in the start and stop services of the EFI_DRIVER_BINDING_PROTOCOL.

11.1 EFI Driver Binding Protocol

This section provides a detailed description of the EFI_DRIVER_BINDING_PROTOCOL. This protocol is produced by every driver that follows the UEFI Driver Model, and it is the central component that allows drivers and controllers to be managed. It provides a service to test if a specific controller is supported by a driver, a service to start managing a controller, and a service to stop managing a controller. These services apply equally to drivers for both bus controllers and device controllers.

EFI_DRIVER_BINDING_PROTOCOL

Summary

Provides the services required to determine if a driver supports a given controller. If a controller is supported, then it also provides routines to start and stop the controller.

GUID

```
#define EFI_DRIVER_BINDING_PROTOCOL_GUID \ 
{0x18A031AB,0xB443,0x4D1A,\ 
{0xA5,0xC0,0x0C,0x09,0x26,0x1E,0x71}}
```

Protocol Interface Structure

```
typedef struct _EFI_DRIVER_BINDING_PROTOCOL {
    EFI_DRIVER_BINDING_PROTOCOL_SUPPORTED Supported;
    EFI_DRIVER_BINDING_PROTOCOL_START Start;
    EFI_DRIVER_BINDING_PROTOCOL_STOP Stop;
    UINT32 Version;
    EFI_HANDLE ImageHandle;
    EFI_HANDLE DriverBindingHandle;
} EFI_DRIVER_BINDING_PROTOCOL;
```
Parameters

**Supported**
Tests to see if this driver supports a given controller. This service is called by the EFI boot service `EFI_BOOT_SERVICES.ConnectController()`. In order to make drivers as small as possible, there are a few calling restrictions for this service. `ConnectController()` must follow these calling restrictions. If any other agent wishes to call `Supported()` it must also follow these calling restrictions. See the `Supported()` function description.

**Start**
Starts a controller using this driver. This service is called by the EFI boot service `ConnectController()`. In order to make drivers as small as possible, there are a few calling restrictions for this service. `ConnectController()` must follow these calling restrictions. If any other agent wishes to call `Start()` it must also follow these calling restrictions. See the `Start()` function description.

**Stop**
Stops a controller using this driver. This service is called by the EFI boot service `EFI_BOOT_SERVICES.DisconnectController()`. In order to make drivers as small as possible, there are a few calling restrictions for this service. `DisconnectController()` must follow these calling restrictions. If any other agent wishes to call `Stop()` it must also follow these calling restrictions. See the `Stop()` function description.

**Version**
The version number of the UEFI driver that produced the `EFI_DRIVER_BINDING_PROTOCOL`. This field is used by the EFI boot service `ConnectController()` to determine the order that driver’s `Supported()` service will be used when a controller needs to be started. EFI Driver Binding Protocol instances with higher `Version` values will be used before ones with lower `Version` values. The `Version` values of `0x0-0x0f` and `0xffffffff-0xffffffff` are reserved for platform/OEM specific drivers. The `Version` values of `0x10-0xffffffef` are reserved for IHV-developed drivers.

**ImageHandle**
The image handle of the UEFI driver that produced this instance of the `EFI_DRIVER_BINDING_PROTOCOL`.

**DriverBindingHandle**
The handle on which this instance of the `EFI_DRIVER_BINDING_PROTOCOL` is installed. In most cases, this is the same handle as `ImageHandle`. However, for UEFI drivers that produce more than one instance of the `EFI_DRIVER_BINDING_PROTOCOL`, this value may not be the same as `ImageHandle`.

Description

The `EFI_DRIVER_BINDING_PROTOCOL` provides a service to determine if a driver supports a given controller. If a controller is supported, then it also provides services to start and stop the controller. All UEFI drivers are required to be reentrant so they can manage one or more controllers. This requires that drivers not use global variables to store device context. Instead, they must allocate a separate context structure per controller that the driver is managing. Bus drivers must support starting and stopping the same bus multiple times, and they must also support starting and stopping all of their children, or just a subset of their children.
EFI_DRIVER_BINDING_PROTOCOL.Supported()

Summary
Tests to see if this driver supports a given controller. If a child device is provided, it further tests to see if this driver supports creating a handle for the specified child device.

Prototype

typedef
EFI_STATUS
(EIFIAPI *EFI_DRIVER_BINDING_PROTOCOL_SUPPORTED) (  
    IN EFI_DRIVER_BINDING_PROTOCOL *This,
    IN EFI_HANDLE ControllerHandle,
    IN EFI_DEVICE_PATH_PROTOCOL *RemainingDevicePath OPTIONAL
);

Parameters

This
A pointer to the EFI_DRIVER_BINDING_PROTOCOL instance.

ControllerHandle
The handle of the controller to test. This handle must support a protocol interface that supplies an I/O abstraction to the driver. Sometimes just the presence of this I/O abstraction is enough for the driver to determine if it supports ControllerHandle. Sometimes, the driver may use the services of the I/O abstraction to determine if this driver supports ControllerHandle.

RemainingDevicePath
A pointer to the remaining portion of a device path. For bus drivers, if this parameter is not NULL, then the bus driver must determine if the bus controller specified by ControllerHandle and the child controller specified by RemainingDevicePath are both supported by this bus driver.

Description
This function checks to see if the driver specified by This supports the device specified by ControllerHandle. Drivers will typically use the device path attached to ControllerHandle and/or the services from the bus I/O abstraction attached to ControllerHandle to determine if the driver supports ControllerHandle. This function may be called many times during platform initialization. In order to reduce boot times, the tests performed by this function must be very small, and take as little time as possible to execute. This function must not change the state of any hardware devices, and this function must be aware that the device specified by ControllerHandle may already be managed by the same driver or a different driver. This function must match its calls to EFI_BOOT_SERVICES.AllocatePages() with EFI_BOOT_SERVICES.FreePages(), EFI_BOOT_SERVICES.AllocatePool() with EFI_BOOT_SERVICES.FreePool(), and EFI_BOOT_SERVICES.OpenProtocol() with EFI_BOOT_SERVICES.CloseProtocol(). Since ControllerHandle may have been previously started by the same driver, if a protocol is already in the opened state, then it must not be closed with CloseProtocol(). This is required to guarantee the state of ControllerHandle is not modified by this function.
If any of the protocol interfaces on the device specified by `ControllerHandle` that are required by the driver specified by `This` are already open for exclusive access by a different driver or application, then `EFI_ACCESS_DENIED` is returned.

If any of the protocol interfaces on the device specified by `ControllerHandle` that are required by the driver specified by `This` are already opened by the same driver, then `EFI_ALREADY_STARTED` is returned. However, if the driver specified by `This` is a bus driver, then it is not an error, and the bus driver should continue with its test of `ControllerHandle` and `RemainingDevicePath`. This allows a bus driver to create one child handle on the first call to `Supported()` and `Start()`, and create additional child handles on additional calls to `Supported()` and `Start()`. This also allows a bus driver to create no child handle on the first call to `Supported()` and `Start()` by specifying an End of Device Path Node `RemainingDevicePath`, and create additional child handles on additional calls to `Supported()` and `Start()`.

If `ControllerHandle` is not supported by `This`, then `EFI_UNSUPPORTED` is returned.

If `This` is a bus driver that creates child handles with an `EFI_DEVICE_PATH_PROTOCOL`, then `ControllerHandle` must support the `EFI_DEVICE_PATH_PROTOCOL`. If it does not, then `EFI_UNSUPPORTED` is returned.

If `ControllerHandle` is supported by `This`, and `This` is a device driver, then `EFI_SUCCESS` is returned.

If `ControllerHandle` is supported by `This`, and `This` is a bus driver, and `RemainingDevicePath` is `NULL` or the first Device Path Node is the End of Device Path Node, then `EFI_SUCCESS` is returned.

If `ControllerHandle` is supported by `This`, and `This` is a bus driver, and `RemainingDevicePath` is not `NULL`, then `RemainingDevicePath` must be analyzed. If the first node of `RemainingDevicePath` is the End of Device Path Node or an EFI Device Path node that the bus driver recognizes and supports, then `EFI_SUCCESS` is returned. Otherwise, `EFI_UNSUPPORTED` is returned.

The `Supported()` function is designed to be invoked from the EFI boot service `EFI_BOOT_SERVICES.ConnectController()`. As a result, much of the error checking on the parameters to `Supported()` has been moved into this common boot service. It is legal to call `Supported()` from other locations, but the following calling restrictions must be followed or the system behavior will not be deterministic.

`ControllerHandle` must be a valid `EFI_HANDLE`. If `RemainingDevicePath` is not `NULL`, then it must be a pointer to a naturally aligned `EFI_DEVICE_PATH_PROTOCOL`.
Status Codes Returned

<table>
<thead>
<tr>
<th>Code</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>EFI_SUCCESS</td>
<td>The device specified by ControllerHandle and RemainingDevicePath is supported by the driver specified by This.</td>
</tr>
<tr>
<td>EFI_ALREADY_STARTED</td>
<td>The device specified by ControllerHandle and RemainingDevicePath is already being managed by the driver specified by This.</td>
</tr>
<tr>
<td>EFI_ACCESS_DENIED</td>
<td>The device specified by ControllerHandle and RemainingDevicePath is already being managed by a different driver or an application that requires exclusive access.</td>
</tr>
<tr>
<td>EFI_UNSUPPORTED</td>
<td>The device specified by ControllerHandle and RemainingDevicePath is not supported by the driver specified by This.</td>
</tr>
</tbody>
</table>

Examples

```c
extern EFI_GUID gEfiDriverBindingProtocolGuid;
EFI_HANDLE DriverImageHandle;
EFI_HANDLE ControllerHandle;
EFI_DRIVER_BINDING_PROTOCOL *DriverBinding;
EFI_DEVICE_PATH_PROTOCOL *RemainingDevicePath;

// Use the DriverImageHandle to get the Driver Binding Protocol instance
//
Status = gBS->OpenProtocol (  
    DriverImageHandle,  
    &gEfiDriverBindingProtocolGuid,  
    &DriverBinding,  
    DriverImageHandle,  
    NULL,  
    EFI_OPEN_PROTOCOL_GET_PROTOCOL  
);  
if (EFI_ERROR (Status)) {
    return Status;
}

// EXAMPLE #1
// Use the Driver Binding Protocol instance to test to see if the driver specified by DriverImageHandle supports the controller specified by ControllerHandle
//
Status = DriverBinding->Supported (  
    DriverBinding,  
    ControllerHandle,  
    NULL  
);  
return Status;

// EXAMPLE #2
// The RemainingDevicePath parameter can be used to initialize only the minimum devices required to boot. For example, maybe we only
// want to initialize 1 hard disk on a SCSI channel. If DriverImageHandle
// is a SCSI Bus Driver, and ControllerHandle is a SCSI Controller, and
// we only want to create a child handle for PUN=3 and LUN=0, then the
// RemainingDevicePath would be SCSI(3,0)/END. The following example
// would return EFI_SUCCESS if the SCSI driver supports creating the
// child handle for PUN=3, LUN=0. Otherwise it would return an error.
//
// Status = DriverBinding->Supported(
//          DriverBinding,
//          ControllerHandle,
//          RemainingDevicePath
//         );
//
// return Status;

Pseudo Code

Listed below are the algorithms for the Supported() function for three different types of drivers. How the Start() function of a driver is implemented can affect how the Supported() function is implemented. All of the services in the EFI_DRIVER_BINDING_PROTOCOL need to work together to make sure that all resources opened or allocated in Supported() and Start() are released in Stop().

The first algorithm is a simple device driver that does not create any additional handles. It only attaches one or more protocols to an existing handle. The second is a bus driver that always creates all of its child handles on the first call to Start(). The third is a more advanced bus driver that can either create one child handles at a time on successive calls to Start(), or it can create all of its child handles or all of the remaining child handles in a single call to Start().

Device Driver:

1. Ignore the parameter RemainingDevicePath.
2. Open all required protocols with EFI_BOOT_SERVICES.OpenProtocol(). A standard driver should use an Attribute of EFI_OPEN_PROTOCOL_BY_DRIVER. If this driver needs exclusive access to a protocol interface, and it requires any drivers that may be using the protocol interface to disconnect, then the driver should use an Attribute of EFI_OPEN_PROTOCOL_BY_DRIVER | EFI_OPEN_PROTOCOL_EXCLUSIVE.
3. If any of the calls to OpenProtocol() in (2) returned an error, then close all of the protocols opened in (2) with EFI_BOOT_SERVICES.CloseProtocol(), and return the status code from the call to OpenProtocol() that returned an error.
4. Use the protocol instances opened in (2) to test to see if this driver supports the controller. Sometimes, just the presence of the protocols is enough of a test. Other times, the services of the protocols opened in (2) are used to further check the identity of the controller. If any of these tests fails, then close all the protocols opened in (2) with CloseProtocol() and return EFI_UNSUPPORTED.
5. Close all protocols opened in (2) with CloseProtocol().
6. Return EFI_SUCCESS.

Bus Driver that creates all of its child handles on the first call to Start():

1. Check the contents of the first Device Path Node of RemainingDevicePath to make sure it is the End of Device Path Node or a legal Device Path Node for this bus driver’s children. If it is not, then return EFI_UNSUPPORTED.
2. Open all required protocols with \texttt{EFI\_BOOT\_SERVICES.\texttt{OpenProtocol}()}. A standard driver should use an \texttt{Attribute} of \texttt{EFI\_OPEN\_PROTOCOL\_BY\_DRIVER}. If this driver needs exclusive access to a protocol interface, and it requires any drivers that may be using the protocol interface to disconnect, then the driver should use an \texttt{Attribute} of \texttt{EFI\_OPEN\_PROTOCOL\_BY\_DRIVER} \texttt{EFI\_OPEN\_PROTOCOL\_EXCLUSIVE}.

3. If any of the calls to \texttt{OpenProtocol()} in (2) returned an error, then close all of the protocols opened in (2) with \texttt{EFI\_BOOT\_SERVICES.\texttt{CloseProtocol}()}, and return the status code from the call to \texttt{OpenProtocol()} that returned an error.

4. Use the protocol instances opened in (2) to test to see if this driver supports the controller. Sometimes, just the presence of the protocols is enough of a test. Other times, the services of the protocols opened in (2) are used to further check the identity of the controller. If any of these tests fails, then close all the protocols opened in (2) with \texttt{CloseProtocol()} and return \texttt{EFI\_UNSUPPORTED}.

5. Close all protocols opened in (2) with \texttt{CloseProtocol}().

6. Return \texttt{EFI\_SUCCESS}.

**Bus Driver that is able to create all or one of its child handles on each call to \texttt{Start}():**

1. Check the contents of the first Device Path Node of \texttt{RemainingDevicePath} to make sure it is the End of Device Path Node or a legal Device Path Node for this bus driver’s children. If it is not, then return \texttt{EFI\_UNSUPPORTED}.

2. Open all required protocols with \texttt{OpenProtocol()}. A standard driver should use an \texttt{Attribute} of \texttt{EFI\_OPEN\_PROTOCOL\_BY\_DRIVER}. If this driver needs exclusive access to a protocol interface, and it requires any drivers that may be using the protocol interface to disconnect, then the driver should use an \texttt{Attribute} of \texttt{EFI\_OPEN\_PROTOCOL\_BY\_DRIVER} \texttt{EFI\_OPEN\_PROTOCOL\_EXCLUSIVE}.

3. If any of the calls to \texttt{OpenProtocol()} in (2) failed with an error other than \texttt{EFI\_ALREADY\_STARTED}, then close all of the protocols opened in (2) that did not return \texttt{EFI\_ALREADY\_STARTED} with \texttt{CloseProtocol()}, and return the status code from the \texttt{OpenProtocol()} call that returned an error.

4. Use the protocol instances opened in (2) to test to see if this driver supports the controller. Sometimes, just the presence of the protocols is enough of a test. Other times, the services of the protocols opened in (2) are used to further check the identity of the controller. If any of these tests fails, then close all the protocols opened in (2) that did not return \texttt{EFI\_ALREADY\_STARTED} with \texttt{CloseProtocol()} and return \texttt{EFI\_UNSUPPORTED}.

5. Close all protocols opened in (2) that did not return \texttt{EFI\_ALREADY\_STARTED} with \texttt{CloseProtocol}().

6. Return \texttt{EFI\_SUCCESS}.

Listed below is sample code of the \texttt{Supported()} function of device driver for a device on the XYZ bus. The XYZ bus is abstracted with the \texttt{EFI\_XYZ\_IO\_PROTOCOL}. Just the presence of the \texttt{EFI\_XYZ\_IO\_PROTOCOL} on \texttt{ControllerHandle} is enough to determine if this driver supports \texttt{ControllerHandle}. The \texttt{gBS} variable is initialized in this driver’s entry point. See Section 4.

```c
extern EFI_GUID                 gEfiXyzIoProtocol;
EFI_BOOT_SERVICES               *gBS;
EFI_STATUS
AbcSupported ( 
```
IN EFI_DRIVER_BINDING_PROTOCOL *This,
IN EFI_HANDLE ControllerHandle,
IN EFI_DEVICE_PATH_PROTOCOL *RemainingDevicePath OPTIONAL
)

{
  EFI_STATUS Status;
  EFI_XYZ_IO_PROTOCOL *XyzIo;

  Status = gBS->OpenProtocol (  
    ControllerHandle,
    &gEfiXyzIoProtocol,
    &XyzIo,
    This->DriverBindingHandle,
    ControllerHandle,
    EFI_OPEN_PROTOCOL_BY_DRIVER  
  );
  if (EFI_ERROR (Status)) {
    return Status;
  }

  gBS->CloseProtocol (  
    ControllerHandle,
    &gEfiXyzIoProtocol,
    This->DriverBindingHandle,
    ControllerHandle
  );

  return EFI_SUCCESS;
}

EFI_DRIVER_BINDING_PROTOCOL.Start()

Summary

Starts a device controller or a bus controller. The Start() and Stop() services of the EFI_DRIVER_BINDING_PROTOCOL mirror each other.

Prototype

typedef
  EFI_STATUS
  (EFIAPI *EFI_DRIVER_BINDING_PROTOCOL_START) (  
    IN EFI_DRIVER_BINDING_PROTOCOL *This,
    IN EFI_HANDLE ControllerHandle,
    IN EFI_DEVICE_PATH_PROTOCOL *RemainingDevicePath OPTIONAL
  );

Parameters

  This  A pointer to the EFI_DRIVER_BINDING_PROTOCOL instance.
  ControllerHandle  The handle of the controller to start. This handle must support a protocol interface that supplies an I/O abstraction to the driver.
**RemainingDevicePath**

A pointer to the remaining portion of a device path. For a bus driver, if this parameter is `NULL`, then handles for all the children of `Controller` are created by this driver.

If this parameter is not `NULL` and the first Device Path Node is not the End of Device Path Node, then only the handle for the child device specified by the first Device Path Node of `RemainingDevicePath` is created by this driver.

If the first Device Path Node of `RemainingDevicePath` is the End of Device Path Node, no child handle is created by this driver.

**Description**

This function starts the device specified by `Controller` with the driver specified by `This`. Whatever resources are allocated in `Start()` must be freed in `Stop()`. For example, every `EFI_BOOT_SERVICES.AllocatePool()`, `EFI_BOOT_SERVICES.AllocatePages()`, `EFI_BOOT_SERVICES.OpenProtocol()`, and `EFI_BOOT_SERVICES.InstallProtocolInterface()` in `Start()` must be matched with a `EFI_BOOT_SERVICES.FreePool()`, `EFI_BOOT_SERVICES.FreePages()`, `EFI_BOOT_SERVICES.CloseProtocol()`, and `EFI_BOOT_SERVICES.UninstallProtocolInterface()` in `Stop()`.

If `Controller` is started, then `EFI_SUCCESS` is returned.

If `Controller` could not be started, but can potentially be repaired with configuration or repair operations using the `EFI_DRIVER_HEALTH_PROTOCOL` and this driver produced an instance of the `EFI_DRIVER_HEALTH_PROTOCOL` for `Controller`, then return `EFI_SUCCESS`.

If `Controller` cannot be started due to a device error and the driver does not produce the `EFI_DRIVER_HEALTH_PROTOCOL` for `Controller`, then return `EFI_DEVICE_ERROR`.

If the driver does not support `Controller` then `EFI_DEVICE_ERROR` is returned. This condition will only be met if `Supported()` returns `EFI_SUCCESS` and a more extensive supported check in `Start()` fails.

If there are not enough resources to start the device or bus specified by `Controller`, then `EFI_OUT_OF_RESOURCES` is returned.

If the driver specified by `This` is a device driver, then `RemainingDevicePath` is ignored.

If the driver specified by `This` is a bus driver, and `RemainingDevicePath` is `NULL`, then all of the children of `Controller` are discovered and enumerated, and a device handle is created for each child.

If the driver specified by `This` is a bus driver, and `RemainingDevicePath` is not `NULL` and begins with the End of Device Path node, then the driver must not enumerate any of the children of `Controller` nor create any child device handle. Only the controller initialization should be performed. If the driver implements `EFI_DRIVER_DIAGNOSTICS2_PROTOCOL`, `EFI_COMPONENT_NAME2_PROTOCOL`, `EFI_SERVICE_BINDING_PROTOCOL`, `EFI_DRIVER_FAMILY_OVERRIDE_PROTOCOL`, or `EFI_DRIVER_HEALTH_PROTOCOL`, the driver still should install the implemented protocols. If the driver supports `EFI_PLATFORM_TO_DRIVER_CONFIGURATION_PROTOCOL`, the driver still should retrieve and process the configuration information.

If the driver specified by `This` is a bus driver that is capable of creating one child handle at a time and `RemainingDevicePath` is not `NULL` and does not begin with the End of Device Path node, then an attempt
is made to create the device handle for the child device specified by `RemainingDevicePath`. Depending on the bus type, all of the child devices may need to be discovered and enumerated, but at most only the device handle for the one child specified by `RemainingDevicePath` shall be created.

The `Start()` function is designed to be invoked from the EFI boot service `EFI_BOOT_SERVICES.ConnectController()`. As a result, much of the error checking on the parameters to `Start()` has been moved into this common boot service. It is legal to call `Start()` from other locations, but the following calling restrictions must be followed or the system behavior will not be deterministic.

- `ControllerHandle` must be a valid `EFI_HANDLE`.
- If `RemainingDevicePath` is not `NULL`, then it must be a pointer to a naturally aligned `EFI_DEVICE_PATH_PROTOCOL`.
- Prior to calling `Start()`, the `Supported()` function for the driver specified by `This` must have been called with the same calling parameters, and `Supported()` must have returned `EFI_SUCCESS`.

### Status Codes Returned

<table>
<thead>
<tr>
<th>Status Code</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>EFI_SUCCESS</code></td>
<td>The device was started.</td>
</tr>
<tr>
<td><code>EFI_SUCCESS</code></td>
<td>The device could not be started because the device needs to be configured by the user or requires a repair operation, and the driver produced the Driver Health Protocol that will return the required configuration and repair operations for this device.</td>
</tr>
<tr>
<td><code>EFI_DEVICE_ERROR</code></td>
<td>The driver does not produce the Driver Health Protocol and the device could not be started due to a device error.</td>
</tr>
<tr>
<td><code>EFI_DEVICE_ERROR</code></td>
<td>The driver produces the Driver Health Protocol, and the driver does not support the device.</td>
</tr>
<tr>
<td><code>EFI_OUT_OF_RESOURCES</code></td>
<td>The request could not be completed due to a lack of resources.</td>
</tr>
</tbody>
</table>

### Examples

```c
extern EFI_GUID             gEfiDriverBindingProtocolGuid;
EFI_HANDLE                  DriverImageHandle;
EFI_HANDLE                  ControllerHandle;
EFI_DRIVER_BINDING_PROTOCOL *DriverBinding;
EFI_DEVICE_PATH_PROTOCOL    *RemainingDevicePath;

// Use the DriverImageHandle to get the Driver Binding Protocol instance
//
Status = gBS->OpenProtocol (  
      DriverImageHandle,  
      &gEfiDriverBindingProtocolGuid,  
      &DriverBinding,  
      DriverImageHandle,  
      NULL,  
      EFI_OPEN_PROTOCOL_GET_PROTOCOL  
);  
```

if (EFI_ERROR (Status)) {
    return Status;
}

// EXAMPLE #1

// Use the Driver Binding Protocol instance to test to see if the
// driver specified by DriverImageHandle supports the controller
// specified by ControllerHandle

Status = DriverBinding->Supported (  
    DriverBinding,  
    ControllerHandle,  
    NULL  
);
if (!EFI_ERROR (Status)) {
    Status = DriverBinding->Start (  
        DriverBinding,  
        ControllerHandle,  
        NULL  
    );
}
return Status;

// EXAMPLE #2

// The RemainingDevicePath parameter can be used to initialize only
// the minimum devices required to boot. For example, maybe we only
// want to initialize 1 hard disk on a SCSI channel. If DriverImageHandle
// is a SCSI Bus Driver, and ControllerHandle is a SCSI Controller, and
// we only want to create a child handle for PUN=3 and LUN=0, then the
// RemainingDevicePath would be SCSI(3,0)/END. The following example
// would return EFI_SUCCESS if the SCSI driver supports creating the
// child handle for PUN=3, LUN=0. Otherwise it would return an error.

Status = DriverBinding->Supported (  
    DriverBinding,  
    ControllerHandle,  
    RemainingDevicePath  
);
if (!EFI_ERROR (Status)) {
    Status = DriverBinding->Start (  
        DriverBinding,  
        ControllerHandle,  
        RemainingDevicePath  
    );
}
return Status;
Pseudo Code

Listed below are the algorithms for the `Start()` function for three different types of drivers. How the `Start()` function of a driver is implemented can affect how the `Supported()` function is implemented. All of the services in the `EFI_DRIVER_BINDING_PROTOCOL` need to work together to make sure that all resources opened or allocated in `Supported()` and `Start()` are released in `Stop()`.

The first algorithm is a simple device driver that does not create any additional handles. It only attaches one or more protocols to an existing handle. The second is a simple bus driver that always creates all of its child handles on the first call to `Start()`. It does not attach any additional protocols to the handle for the bus controller. The third is a more advanced bus driver that can either create one child handles at a time on successive calls to `Start()`, or it can create all of its child handles or all of the remaining child handles in a single call to `Start()`. Once again, it does not attach any additional protocols to the handle for the bus controller.

**Device Driver:**

1. Ignore the parameter `RemainingDevicePath`.
2. Open all required protocols with `EFI_BOOT_SERVICES.OpenProtocol()`. A standard driver should use an `Attribute` of `EFI_OPEN_PROTOCOL_BY_DRIVER`. If this driver needs exclusive access to a protocol interface, and it requires any drivers that may be using the protocol interface to disconnect, then the driver should use an `Attribute` of `EFI_OPEN_PROTOCOL_BY_DRIVER | EFI_OPEN_PROTOCOL_EXCLUSIVE`. It must use the same `Attribute` value that was used in `Supported()`.
3. If any of the calls to `OpenProtocol()` in (2) returned an error, then close all of the protocols opened in (2) with `EFI_BOOT_SERVICES.CloseProtocol()`, and return the status code from the call to `OpenProtocol()` that returned an error.
4. Initialize the device specified by `ControllerHandle`. If the driver does not support the device specified by `ControllerHandle`, then close all of the protocols opened in (2) with `CloseProtocol()`, and return `EFI_DEVICE_ERROR`. If the driver does support the device specified by `ControllerHandle` and an error is detected, and that error can not be resolved with the `EFI_DRIVER_HEALTH_PROTOCOL`, then close all of the protocols opened in (2) with `CloseProtocol()`, and return `EFI_DEVICE_ERROR`. If the driver does support the device specified by `ControllerHandle` and an error is detected, and that error can be resolved with the `EFI_DRIVER_HEALTH_PROTOCOL`, then produce the `EFI_DRIVER_HEALTH_PROTOCOL` for `ControllerHandle` and make sure `EFI_SUCCESS` is returned from `Start()`. In this case, depending on the type of error detected, not all of the following steps may be completed.
5. Allocate and initialize all of the data structures that this driver requires to manage the device specified by `ControllerHandle`. This would include space for public protocols and space for any additional private data structures that are related to `ControllerHandle`. If an error occurs allocating the resources, then close all of the protocols opened in (2) with `CloseProtocol()`, and return `EFI_OUT_OF_RESOURCES`.
6. Install all the new protocol interfaces onto `ControllerHandle` using `EFI_BOOT_SERVICES.InstallMultipleProtocolInterfaces()`. If an error occurs, close all of the protocols opened in (1) with `CloseProtocol()`, and return the error from `InstallMultipleProtocolInterfaces()`.
7. Return `EFI_SUCCESS`. 
Bus Driver that creates all of its child handles on the first call to Start():

1. Ignore the parameter RemainingDevicePath. with the exception that if the first Device Path Node is the End of Device Path Node, skip steps 5-8.

2. Open all required protocols with EFI_BOOT_SERVICES.OpenProtocol(). A standard driver should use an Attribute of EFI_OPEN_PROTOCOL_BY_DRIVER. If this driver needs exclusive access to a protocol interface, and it requires any drivers that may be using the protocol interface to disconnect, then the driver should use an Attribute of EFI_OPEN_PROTOCOL_BY_DRIVER | EFI_OPEN_PROTOCOL_EXCLUSIVE. It must use the same Attribute value that was used in Supported().

3. If any of the calls to OpenProtocol() in (2) returned an error, then close all of the protocols opened in (2) with EFI_BOOT_SERVICES.CloseProtocol(), and return the status code from the call to OpenProtocol() that returned an error.

4. Initialize the device specified by ControllerHandle. If the driver does not support the device specified by ControllerHandle, then close all of the protocols opened in (2) with CloseProtocol(), and return EFI_DEVICE_ERROR. If the driver does support the device specified by ControllerHandle and an error is detected, and that error can not be resolved with the EFI_DRIVER_HEALTH_PROTOCOL, then close all of the protocols opened in (2) with CloseProtocol(), and return EFI_DEVICE_ERROR. If the driver does support the device specified by ControllerHandle and an error is detected, and that error can be resolved with the EFI_DRIVER_HEALTH_PROTOCOL, then produce the EFI_DRIVER_HEALTH_PROTOCOL for ControllerHandle and make sure EFI_SUCCESS is returned from Start(). In this case, depending on the type of error detected, not all of the following steps may be completed.

5. Discover all the child devices of the bus controller specified by ControllerHandle.

6. If the bus requires it, allocate resources to all the child devices of the bus controller specified by ControllerHandle.

7. FOR each child C of ControllerHandle:
   a. Allocate and initialize all of the data structures that this driver requires to manage the child device C. This would include space for public protocols and space for any additional private data structures that are related to the child device C. If an error occurs allocating the resources, then close all of the protocols opened in (2) with CloseProtocol(), and return EFI_OUT_OF_RESOURCES.
   b. If the bus driver creates device paths for the child devices, then create a device path for the child C based upon the device path attached to ControllerHandle.
   c. Initialize the child device C. If an error occurs, close all of the protocols opened in (2) with CloseProtocol(), and return EFI_DEVICE_ERROR.
   d. Create a new handle for C, and install the protocol interfaces for child device C using EFI_BOOT_SERVICES.InstallMultipleProtocolInterfaces(). This may include the EFI_DEVICE_PATH_PROTOCOL.
   e. Call OpenProtocol() on behalf of the child C with an Attribute of EFI_OPEN_PROTOCOL_BY_CHILD_CONTROLLER.

8. END FOR

9. If the bus driver also produces protocols on ControllerHandle, then install all the new protocol interfaces onto ControllerHandle using InstallMultipleProtocolInterfaces(). If an error occurs, close all of the protocols
opened in (2) with CloseProtocol(), and return the error from InstallMultipleProtocolInterfaces().

10. Return EFI_SUCCESS.

Bus Driver that is able to create all or one of its child handles on each call to Start():

1. Open all required protocols with EFI_BOOT_SERVICES.OpenProtocol(). A standard driver should use an Attribute of EFI_OPEN_PROTOCOL_BY_DRIVER. If this driver needs exclusive access to a protocol interface, and it requires any drivers that may be using the protocol interface to disconnect, then the driver should use an Attribute of EFI_OPEN_PROTOCOL_BY_DRIVER | EFI_OPEN_PROTOCOL_EXCLUSIVE. It must use the same Attribute value that was used in Supported().

2. If any of the calls to OpenProtocol() in (1) returned an error, then close all of the protocols opened in (1) with EFI_BOOT_SERVICES.CloseProtocol(), and return the status code from the call to OpenProtocol() that returned an error.

3. Initialize the device specified by ControllerHandle. If the driver does not support the device specified by ControllerHandle, then close all of the protocols opened in (1) with CloseProtocol(), and return EFI_DEVICE_ERROR. If the driver does support the device specified by ControllerHandle and an error is detected, and that error cannot be resolved with the EFI_DRIVER_HEALTH_PROTOCOL, then close all of the protocols opened in (1) with CloseProtocol(), and return EFI_DEVICE_ERROR. If the driver does support the device specified by ControllerHandle and an error is detected, and that error can be resolved with the EFI_DRIVER_HEALTH_PROTOCOL, then produce the EFI_DRIVER_HEALTH_PROTOCOL for ControllerHandle and make sure EFI_SUCCESS is returned from Start(). In this case, depending on the type of error detected, not all of the following steps may be completed.

4. IF RemainingDevicePath is not NULL, THEN
   a. C is the child device specified by RemainingDevicePath. If the first Device Path Node is the End of Device Path Node, proceed to step 6.
   b. Allocate and initialize all of the data structures that this driver requires to manage the child device C. This would include space for public protocols and space for any additional private data structures that are related to the child device C. If an error occurs allocating the resources, then close all of the protocols opened in (1) with CloseProtocol(), and return EFI_OUT_OF_RESOURCES.
   c. If the bus driver creates device paths for the child devices, then create a device path for the child C based upon the device path attached to ControllerHandle.
   d. Initialize the child device C.
   e. Create a new handle for C, and install the protocol interfaces for child device C using EFI_BOOT_SERVICES.InstallMultipleProtocolInterfaces(). This may include the EFI_DEVICE_PATH_PROTOCOL.
   f. Call OpenProtocol() on behalf of the child C with an Attribute of EFI_OPEN_PROTOCOL_BY_CHILD_CONTROLLER.

   ELSE
   a. Discover all the child devices of the bus controller specified by ControllerHandle.
   b. If the bus requires it, allocate resources to all the child devices of the bus controller specified by ControllerHandle.
   c. FOR each child C of ControllerHandle
Allocate and initialize all of the data structures that this driver requires to manage the child device C. This would include space for public protocols and space for any additional private data structures that are related to the child device C. If an error occurs allocating the resources, then close all of the protocols opened in (1) with CloseProtocol(), and return EFI_OUT_OF_RESOURCES.

If the bus driver creates device paths for the child devices, then create a device path for the child C based upon the device path attached to ControllerHandle. Initialize the child device C.

Create a new handle for C, and install the protocol interfaces for child device C using InstallMultipleProtocolInterfaces(). This may include the EFI_DEVICE_PATH_PROTOCOL.

Call EFI_BOOT_SERVICES.OpenProtocol() on behalf of the child C with an Attribute of EFI_OPEN_PROTOCOL_BY_CHILD_CONTROLLER.

For END IF

5. END IF

6. If the bus driver also produces protocols on ControllerHandle, then install all the new protocol interfaces onto ControllerHandle using InstallMultipleProtocolInterfaces(). If an error occurs, close all of the protocols opened in (2) with CloseProtocol(), and return the error from InstallMultipleProtocolInterfaces().

7. Return EFI_SUCCESS.

Listed below is sample code of the Start() function of a device driver for a device on the XYZ bus. The XYZ bus is abstracted with the EFI_XYZ_IO_PROTOCOL. This driver does allow the EFI_XYZ_IO_PROTOCOL to be shared with other drivers, and just the presence of the EFI_XYZ_IO_PROTOCOL on ControllerHandle is enough to determine if this driver supports ControllerHandle. This driver installs the EFI_ABC_IO_PROTOCOL on ControllerHandle. The gBS variable is initialized in this driver’s entry point as shown in the UEFI Driver Model examples in Section 1.6.

```c
extern EFI_GUID gEfiXyzIoProtocol;
extern EFI_GUID gEfiAbcIoProtocol;
EFI_BOOT_SERVICES *gBS;

EFI_STATUS AbcStart (  
    IN EFI_DRIVER_BINDING_PROTOCOL *This,
    IN EFI_HANDLE                  ControllerHandle,
    IN EFI_DEVICE_PATH_PROTOCOL    *RemainingDevicePath OPTIONAL  
)
{
    EFI_STATUS Status;
    EFI_XYZ_IO_PROTOCOL *XyzIo;
    EFI_ABC_DEVICE       AbcDevice;

    // Open the Xyz I/O Protocol that this driver consumes
    //
```
Status = gBS->OpenProtocol (ControllerHandle, 
  &gEfiXyzIoProtocol, 
  &XyzIo, 
  This->DriverBindingHandle, 
  ControllerHandle, 
  EFI_OPEN_PROTOCOL_BY_DRIVER);
if (EFI_ERROR (Status)) {
  return Status;
}
// // Allocate and zero a private data structure for the Abc device. // Status = gBS->AllocatePool ( 
  EfiBootServicesData, 
  sizeof (EFI_ABC_DEVICE), 
  &AbcDevice 
); 
if (EFI_ERROR (Status)) {
  goto ErrorExit;
} 
ZeroMem (AbcDevice, sizeof (EFI_ABC_DEVICE));
// // Initialize the contents of the private data structure for the Abc device. // This includes the XyzIo protocol instance and other private data fields // and the EFI_ABC_IO_PROTOCOL instance that will be installed. // AbcDevice->Signature = EFI_ABC_DEVICE_SIGNATURE; 
AbcDevice->XyzIo = XyzIo; 
AbcDevice->PrivateData1 = PrivateValue1; 
AbcDevice->PrivateData2 = PrivateValue2; 
... 
AbcDevice->PrivateDataN = PrivateValueN; 
AbcDevice->AbcIo.Revision = EFI_ABC_IO_PROTOCOL_REVISION; 
AbcDevice->AbcIo.Func1 = AbcIoFunc1; 
AbcDevice->AbcIo.Func2 = AbcIoFunc2; 
... 
AbcDevice->AbcIo.FuncN = AbcIoFuncN; 
AbcDevice->AbcIo.Data1 = Value1; 
AbcDevice->AbcIo.Data2 = Value2; 
... 
AbcDevice->AbcIo.DataN = ValueN;
// // Install protocol interfaces for the ABC I/O device. // Status = gBS->InstallMultipleProtocolInterfaces ( 
  &ControllerHandle, 
  &gEfiAbcIoProtocolGuid, &AbcDevice->AbcIo, 
  ... 
);
NULL
);
if (EFI_ERROR (Status)) {
    goto ErrorExit;
}

return EFI_SUCCESS;

ErrorExit:
    //
    // When there is an error, the private data structures need to be freed and
    // the protocols that were opened need to be closed.
    //
    if (AbcDevice != NULL) {
        gBS->FreePool (AbcDevice);
    }
    gBS->CloseProtocol (
        ControllerHandle,
        &gEfiXyzIoProtocolGuid,
        This->DriverBindingHandle,
        ControllerHandle
    );
    return Status;
}

EFI_DRIVER_BINDING_PROTOCOL.Stop()

Summary
Stops a device controller or a bus controller. The Start() and Stop() services of the
EFI_DRIVER_BINDING_PROTOCOL mirror each other.

Prototype
typedef
    EFI_STATUS
    (EFIAPI *EFI_DRIVER_BINDING_PROTOCOL_STOP) (
        IN EFI_DRIVER_BINDING_PROTOCOL  *This,
        IN EFI_HANDLE                   ControllerHandle,
        IN UINTN                        NumberOfChildren,
        IN EFI_HANDLE                   *ChildHandleBuffer  OPTIONAL
    );

Parameters
    This          A pointer to the EFI_DRIVER_BINDING_PROTOCOL instance. Type
                   EFI_DRIVER_BINDING_PROTOCOL is defined in Section 11.1.
    ControllerHandle   A handle to the device being stopped. The handle must support a
                        bus specific I/O protocol for the driver to use to stop the device.
    NumberOfChildren  The number of child device handles in ChildHandleBuffer.
    ChildHandleBuffer An array of child handles to be freed. May be NULL if
                        NumberOfChildren is 0.
Description

This function performs different operations depending on the parameter `NumberOfChildren`. If `NumberOfChildren` is not zero, then the driver specified by `This` is a bus driver, and it is being requested to free one or more of its child handles specified by `NumberOfChildren` and `ChildHandleBuffer`. If all of the child handles are freed, then `EFI_SUCCESS` is returned. If `NumberOfChildren` is zero, then the driver specified by `This` is either a device driver or a bus driver, and it is being requested to stop the controller specified by `ControllerHandle`. If `ControllerHandle` is stopped, then `EFI_SUCCESS` is returned. In either case, this function is required to undo what was performed in `Start()`. Whatever resources are allocated in `Start()` must be freed in `Stop()`. For example, every `EFI_BOOT_SERVICES.AllocatePool()`, `EFI_BOOT_SERVICES.AllocatePages()`, `EFI_BOOT_SERVICES.InstallProtocolInterface()` in `Start()` must be matched with a `EFI_BOOT_SERVICES.FreePool()`, `EFI_BOOT_SERVICES.FreePages()`, `EFI_BOOT_SERVICES.CloseProtocol()`, and `EFI_BOOT_SERVICES.UninstallProtocolInterface()` in `Stop()`.

If `ControllerHandle` cannot be stopped, then `EFI_DEVICE_ERROR` is returned. If, for some reason, there are not enough resources to stop `ControllerHandle`, then `EFI_OUT_OF_RESOURCES` is returned.

The `Stop()` function is designed to be invoked from the EFI boot service `EFI_BOOT_SERVICES.DisconnectController()`. As a result, much of the error checking on the parameters to `Stop()` has been moved into this common boot service. It is legal to call `Stop()` from other locations, but the following calling restrictions must be followed or the system behavior will not be deterministic.

- `ControllerHandle` must be a valid `EFI_HANDLE` that was used on a previous call to this same driver’s `Start()` function.
- The first `NumberOfChildren` handles of `ChildHandleBuffer` must all be a valid `EFI_HANDLE`. In addition, all of these handles must have been created in this driver’s `Start()` function, and the `Start()` function must have called `EFI_BOOT_SERVICES.OpenProtocol()` on `ControllerHandle` with an `Attribute` of `EFI_OPEN_PROTOCOL_BY_CHILD_CONTROLLER`.

Status Codes Returned

<table>
<thead>
<tr>
<th>Status Code</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>EFI_SUCCESS</td>
<td>The device was stopped.</td>
</tr>
<tr>
<td>EFI_DEVICE_ERROR</td>
<td>The device could not be stopped due to a device error.</td>
</tr>
</tbody>
</table>

Examples

```c
extern EFI_GUID gEfiDriverBindingProtocolGuid;
EFI_HANDLE DriverImageHandle;
EFI_HANDLE ControllerHandle;
EFI_HANDLE ChildHandle;
EFI_DRIVER_BINDING_PROTOCOL *DriverBinding;

// Use the DriverImageHandle to get the Driver Binding Protocol instance
//
```
Status = gBS->OpenProtocol (  
    DriverImageHandle,  
    &gEfiDriverBindingProtocolGuid,  
    &DriverBinding,  
    DriverImageHandle,  
    NULL,  
    EFI_OPEN_PROTOCOL_GET_PROTOCOL  
);  
if (EFI_ERROR (Status)) {  
    return Status;  
}  

// Use the Driver Binding Protocol instance to free the child  
// specified by ChildHandle. Then, use the Driver Binding  
// Protocol to stop ControllerHandle.  
//  
Status = DriverBinding->Stop (  
    DriverBinding,  
    ControllerHandle,  
    1,  
    &ChildHandle  
);  
Status = DriverBinding->Stop (  
    DriverBinding,  
    ControllerHandle,  
    0,  
    NULL  
);  

Pseudo Code  
Device Driver:  
1. Uninstall all the protocols that were installed onto ControllerHandle in Start().  
2. Close all the protocols that were opened on behalf of ControllerHandle in Start().  
3. Free all the structures that were allocated on behalf of ControllerHandle in Start().  
4. Return EFI_SUCCESS.  

Bus Driver that creates all of its child handles on the first call to Start():  
1. IF NumberOfChildren is zero THEN:  
   a Uninstall all the protocols that were installed onto ControllerHandle in Start().  
   b Close all the protocols that were opened on behalf of ControllerHandle in Start().  
   c Free all the structures that were allocated on behalf of ControllerHandle in Start().  
2. ELSE  
   a FOR each child C in ChildHandleBuffer  
      Uninstall all the protocols that were installed onto C in Start().  
      Close all the protocols that were opened on behalf of C in Start().
Free all the structures that were allocated on behalf of C in **Start()**.

3. END IF
4. Return **EFI_SUCCESS**.

Listed below is sample code of the **Stop()** function of a device driver for a device on the XYZ bus. The XYZ bus is abstracted with the **EFI_XYZ_IO_PROTOCOL**. This driver does allow the **EFI_XYZ_IO_PROTOCOL** to be shared with other drivers, and just the presence of the **EFI_XYZ_IO_PROTOCOL** on **ControllerHandle** is enough to determine if this driver supports **ControllerHandle**. This driver installs the **EFI_ABC_IO_PROTOCOL** on **ControllerHandle** in **Start()**. The **gBS** variable is initialized in this driver’s entry point. See **Section 4**.

```c
extern EFI_GUID     gEfiXyzIoProtocol;
extern EFI_GUID     gEfiAbcIoProtocol;
EFI_BOOT_SERVICES    *gBS;

EFI_STATUS
AbcStop ( 
    IN EFI_DRIVER_BINDING_PROTOCOL *This,
    IN EFI_HANDLE                  ControllerHandle
    IN UINTN                       NumberOfChildren,
    IN EFI_HANDLE                 *ChildHandleBuffer OPTIONAL
) {

    EFI_STATUS      Status;
    EFI_ABC_IO        AbcIo;
    EFI_ABC_DEVICE    AbcDevice;

    // Get our context back
    //
    Status = gBS->OpenProtocol ( 
        ControllerHandle, 
        &gEfiAbcIoProtocolGuid, 
        &AbcIo, 
        This->DriverBindingHandle, 
        ControllerHandle, 
        EFI_OPEN_PROTOCOL_GET_PROTOCOL
    );
    if (EFI_ERROR (Status)) {
        return EFI_UNSUPPORTED;
    }

    // Use Containment Record Macro to get AbcDevice structure from 
    // a pointer to the AbcIo structure within the AbcDevice structure. 
    //
    AbcDevice = ABC_IO_PRIVATE_DATA_FROM_THIS (AbcIo);

    // Uninstall the protocol installed in Start() 
    //
    Status = gBS->UninstallMultipleProtocolInterfaces ( 
```
ControllerHandle,
&gEfiAbcIoProtocolGuid, &AbcDevice->AbcIo, NULL);
if (!EFI_ERROR (Status)) {

  // Close the protocol opened in Start()
  //
  Status = gBS->CloseProtocol (ControllerHandle,
        &gEfiXyzIoProtocolGuid, This->DriverBindingHandle,
        ControllerHandle);

  // Free the structure allocated in Start().
  //
  gBS->FreePool (AbcDevice);
}
return Status;
}

11.2 EFI Platform Driver Override Protocol

This section provides a detailed description of the **EFI_PLATFORM_DRIVER_OVERRIDE_PROTOCOL**. This protocol can override the default algorithm for matching drivers to controllers.

**EFI_PLATFORM_DRIVER_OVERRIDE_PROTOCOL**

**Summary**

This protocol matches one or more drivers to a controller. A platform driver produces this protocol, and it is installed on a separate handle. This protocol is used by the **EFI_BOOT_SERVICES.ConnectController()** boot service to select the best driver for a controller. All of the drivers returned by this protocol have a higher precedence than drivers found from an EFI Bus Specific Driver Override Protocol or drivers found from the general UEFI driver Binding search algorithm. If more than one driver is returned by this protocol, then the drivers are returned in order from highest precedence to lowest precedence.
GUID
#define EFI_PLATFORM_DRIVER_OVERRIDE_PROTOCOL_GUID \
{0x6b30c738,0xa391,0x11d4,\} \
{0x9a,0x3b,0x00,0x90,0x27,0x3f,0xc1,0x4d}

Protocol Interface Structure
typedef struct _EFI_PLATFORM_DRIVER_OVERRIDE_PROTOCOL {
  EFI_PLATFORM_DRIVER_OVERRIDE_GET_DRIVER GetDriver;
  EFI_PLATFORM_DRIVER_OVERRIDE_GET_DRIVER_PATH GetDriverPath;
  EFI_PLATFORM_DRIVER_OVERRIDE_DRIVER_LOADED DriverLoaded;
} EFI_PLATFORM_DRIVER_OVERRIDE_PROTOCOL;

Parameters
  GetDriver  Retrieves the image handle of a platform override driver for a controller in the system. See the GetDriver() function description.
  GetDriverPath  Retrieves the device path of a platform override driver for a controller in the system. See the GetDriverPath() function description.
  DriverLoaded  This function is used after a driver has been loaded using a device path returned by GetDriverPath(). This function associates a device path to an image handle, so the image handle can be returned the next time that GetDriver() is called for the same controller. See the DriverLoaded() function description.

Description
The EFI_PLATFORM_DRIVER_OVERRIDE_PROTOCOL is used by the EFI boot service EFI_BOOT_SERVICES.ConnectController() to determine if there is a platform specific driver override for a controller that is about to be started. The bus drivers in a platform will use a bus defined matching algorithm for matching drivers to controllers. This protocol allows the platform to override the bus driver's default driver matching algorithm. This protocol can be used to specify the drivers for on-board devices whose drivers may be in a system ROM not directly associated with the on-board controller, or it can even be used to manage the matching of drivers and controllers in add-in cards. This can be very useful if there are two adapters that are identical except for the revision of the driver in the adapter's ROM. This protocol, along with a platform configuration utility, could specify which of the two drivers to use for each of the adapters.

The drivers that this protocol returns can be either in the form of an image handle or a device path. EFI_BOOT_SERVICES.ConnectController() can only use image handles, so ConnectController() is required to use the GetDriver() service. A different component, such as the Boot Manager, will have to use the GetDriverPath() service to retrieve the list of drivers that need to be loaded from I/O devices. Once a driver has been loaded and started, this same component can use the DriverLoaded() service to associate the device path of a driver with the image handle of the loaded driver. Once this association has been established, the image handle can then be returned by the GetDriver() service the next time it is called by ConnectController().
EFI_PLATFORM_DRIVER_OVERRIDE_PROTOCOL.GetDriver()

Summary
Retrieves the image handle of the platform override driver for a controller in the system.

Prototype

```c
typedef EFI_STATUS (EFIAPI *EFI_PLATFORM_DRIVER_OVERRIDE_GET_DRIVER) (
    IN   EFI_PLATFORM_DRIVER_OVERRIDE_PROTOCOL *This,
    IN   EFI_HANDLE ControllerHandle,
    IN OUT EFI_HANDLE *DriverImageHandle
);
```

Parameters

- **This**
  A pointer to the EFI_PLATFORM_DRIVER_OVERRIDE_PROTOCOL instance.
- **ControllerHandle**
  The device handle of the controller to check if a driver override exists.
- **DriverImageHandle**
  On input, a pointer to the previous driver image handle returned by GetDriver(). On output, a pointer to the next driver image handle. Passing in a NULL, will return the first driver image handle for ControllerHandle.

Description

This function is used to retrieve a driver image handle that is selected in a platform specific manner. The first driver image handle is retrieved by passing in a `DriverImageHandle` value of `NULL`. This will cause the first driver image handle to be returned in `DriverImageHandle`. On each successive call, the previous value of `DriverImageHandle` must be passed in. If a call to this function returns a valid driver image handle, then `EFI_SUCCESS` is returned. This process is repeated until `EFI_NOT_FOUND` is returned. If a `DriverImageHandle` is passed in that was not returned on a prior call to this function, then `EFI_INVALID_PARAMETER` is returned. If `ControllerHandle` is `NULL`, then `EFI_INVALID_PARAMETER` is returned. The first driver image handle has the highest precedence, and the last driver image handle has the lowest precedence. This ordered list of driver image handles is used by the boot service `EFI_BOOT_SERVICES.ConnectController()` to search for the best driver for a controller.
Status Codes Returned

<table>
<thead>
<tr>
<th>Code</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>EFI_SUCCESS</td>
<td>The driver override for ControllerHandle was returned in DriverImageHandle.</td>
</tr>
<tr>
<td>EFI_NOT_FOUND</td>
<td>A driver override for ControllerHandle was not found.</td>
</tr>
<tr>
<td>EFI_INVALID_PARAMETER</td>
<td>The handle specified by ControllerHandle is not a valid handle.</td>
</tr>
<tr>
<td>EFI_INVALID_PARAMETER</td>
<td>DriverImageHandle is not a handle that was returned on a previous call to GetDriver().</td>
</tr>
</tbody>
</table>

\[
\text{EFI\_PLATFORM\_DRIVER\_OVERRRIDE\_PROTOCOL.GetDriverPath()}(\text{This, ControllerHandle, **DriverImagePath})
\]

Summary

Retrieves the device path of the platform override driver for a controller in the system.

Prototype

\[
\text{typedef EFI\_STATUS (EFIAPI *EFI\_PLATFORM\_DRIVER\_OVERRRIDE\_GET\_DRIVER\_PATH)}(\text{IN EFI\_PLATFORM\_DRIVER\_OVERRRIDE\_PROTOCOL *This, IN EFI\_HANDLE ControllerHandle, IN OUT EFI\_DEVICE\_PATH\_PROTOCOL **DriverImagePath})
\]

Parameters

- **This** A pointer to the EFI_PLATFORM_DRIVER_OVERRIDE_PROTOCOL instance.
- **ControllerHandle** The device handle of the controller to check if a driver override exists.
- **DriverImagePath** On input, a pointer to the previous driver device path returned by GetDriverPath(). On output, a pointer to the next driver device path. Passing in a pointer to NULL, will return the first driver device path for ControllerHandle.

Description

This function is used to retrieve a driver device path that is selected in a platform specific manner. The first driver device path is retrieved by passing in a DriverImagePath value that is a pointer to NULL. This will cause the first driver device path to be returned in DriverImagePath. On each successive call, the previous value of DriverImagePath must be passed in. If a call to this function returns a valid driver device path, then EFI_SUCCESS is returned. This process is repeated until EFI_NOT_FOUND is returned. If a DriverImagePath is passed in that was not returned on a prior call to this function, then EFI_INVALID_PARAMETER is returned. If ControllerHandle is NULL, then EFI_INVALID_PARAMETER is returned. The first driver device path has the highest precedence, and the last driver device path has the lowest precedence. This ordered list of driver device paths is used by a platform specific component, such as the EFI Boot Manager, to load and start the platform override drivers by using the EFI boot services EFI_BOOT_SERVICES.LoadImage() and EFI_BOOT_SERVICES.StartImage(). Each time one of these drivers is loaded and started, the DriverLoaded() service is called.
Status Codes Returned

<table>
<thead>
<tr>
<th>Status Code</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>EFI_SUCCESS</td>
<td>The driver override for ControllerHandle was returned in DriverImagePath.</td>
</tr>
<tr>
<td>EFI_UNSUPPORTED</td>
<td>The operation is not supported.</td>
</tr>
<tr>
<td>EFI_NOT_FOUND</td>
<td>A driver override for ControllerHandle was not found.</td>
</tr>
<tr>
<td>EFI_INVALID_PARAMETER</td>
<td>The handle specified by ControllerHandle is not a valid handle.</td>
</tr>
<tr>
<td>EFI_INVALID_PARAMETER</td>
<td>DriverImagePath is not a device path that was returned on a previous call to GetDriverPath().</td>
</tr>
</tbody>
</table>

### EFI_PLATFORM_DRIVER_OVERRIDE_PROTOCOL.DriverLoaded()

#### Summary

Used to associate a driver image handle with a device path that was returned on a prior call to the GetDriverPath() service. This driver image handle will then be available through the GetDriver() service.

#### Prototype

```c
typedef EFI_STATUS (EFIAPI *EFI_PLATFORM_DRIVER_OVERRIDE_DRIVER_LOADED) (  
  IN EFI_PLATFORM_DRIVER_OVERRIDE_PROTOCOL *This,  
  IN EFI_HANDLE ControllerHandle,  
  IN EFI_DEVICE_PATH_PROTOCOL *DriverImagePath,  
  IN EFI_HANDLE DriverImageHandle  
);
```

#### Parameters

- **This**
  - A pointer to the EFI_PLATFORM_DRIVER_OVERRIDE_PROTOCOL instance.

- **ControllerHandle**
  - The device handle of a controller. This must match the controller handle that was used in a prior call to GetDriver() to retrieve DriverImagePath.

- **DriverImagePath**
  - A pointer to the driver device path that was returned in a prior call to GetDriverPath().

- **DriverImageHandle**
  - The driver image handle that was returned by EFI_BOOT_SERVICES.LoadImage() when the driver specified by DriverImagePath was loaded into memory.

#### Description

This function associates the image handle specified by DriverImageHandle with the device path of a driver specified by DriverImagePath. DriverImagePath must be a value that was returned on a prior call to GetDriverPath() for the controller specified by ControllerHandle. Once this association has been established, then the service GetDriver() must return DriverImageHandle as one of the override drivers for the controller specified by ControllerHandle.
If the association between the image handle specified by `DriverImageHandle` and the device path specified by `DriverImagePath` is established for the controller specified by `ControllerHandle`, then `EFI_SUCCESS` is returned. If `ControllerHandle` is `NULL`, or `DriverImagePath` is not a valid device path, or `DriverImageHandle` is `NULL`, then `EFI_INVALID_PARAMETER` is returned. If `DriverImagePath` is not a device path that was returned on a prior call to `GetDriver()` for the controller specified by `ControllerHandle`, then `EFI_NOT_FOUND` is returned.

### Status Codes Returned

<table>
<thead>
<tr>
<th>Status Code</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>EFI_SUCCESS</td>
<td>The association between <code>DriverImagePath</code> and <code>DriverImageHandle</code> was established for the controller specified by <code>ControllerHandle</code>.</td>
</tr>
<tr>
<td>EFI_UNSUPPORTED</td>
<td>The operation is not supported.</td>
</tr>
<tr>
<td>EFI_NOT_FOUND</td>
<td><code>DriverImagePath</code> is not a device path that was returned on a prior call to <code>GetDriverPath()</code> for the controller specified by <code>ControllerHandle</code>.</td>
</tr>
<tr>
<td>EFI_INVALID_PARAMETER</td>
<td><code>ControllerHandle</code> is not a valid device handle.</td>
</tr>
<tr>
<td>EFI_INVALID_PARAMETER</td>
<td><code>DriverImagePath</code> is not a valid device path.</td>
</tr>
<tr>
<td>EFI_INVALID_PARAMETER</td>
<td><code>DriverImageHandle</code> is not a valid image handle.</td>
</tr>
</tbody>
</table>

## 11.3 EFI Bus Specific Driver Override Protocol

This section provides a detailed description of the `EFI_BUS_SPECIFIC_DRIVER_OVERRIDE_PROTOCOL`. Bus drivers that have a bus specific algorithm for matching drivers to controllers are required to produce this protocol for each controller. For example, a PCI Bus Driver will produce an instance of this protocol for every PCI controller that has a PCI option ROM that contains one or more UEFI drivers. The protocol instance is attached to the handle of the PCI controller.

### EFI_BUS_SPECIFIC_DRIVER_OVERRIDE_PROTOCOL

#### Summary

This protocol matches one or more drivers to a controller. This protocol is produced by a bus driver, and it is installed on the child handles of buses that require a bus specific algorithm for matching drivers to controllers. This protocol is used by the `EFI_BOOT_SERVICES.ConnectController()` boot service to select the best driver for a controller. All of the drivers returned by this protocol have a higher precedence than drivers found in the general UEFI Driver Binding search algorithm, but a lower precedence than those drivers returned by the EFI Platform Driver Override Protocol. If more than one driver image handle is returned by this protocol, then the drivers image handles are returned in order from highest precedence to lowest precedence.
GUID

```c
#define EFI_BUS_SPECIFIC_DRIVER_OVERRIDE_PROTOCOL_GUID \
{0x3bc1b285,0x8a15,0x4a82,\ 
 {0xaa,0xbf,0x4d,0x7d,0x13,0xfb,0x32,0x65}}
```

Protocol Interface Structure

```c
typedef struct EFI_BUS_SPECIFIC_DRIVER_OVERRIDE_PROTOCOL {
    EFI_BUS_SPECIFIC_DRIVER_OVERRIDE_GET_DRIVER GetDriver;
} EFI_BUS_SPECIFIC_DRIVER_OVERRIDE_PROTOCOL;
```

Parameters

- **GetDriver**
  Uses a bus specific algorithm to retrieve a driver image handle for a controller. See the `GetDriver()` function description.

Description

The **EFI_BUS_SPECIFIC_DRIVER_OVERRIDE_PROTOCOL** provides a mechanism for bus drivers to override the default driver selection performed by the `ConnectController()` boot service. This protocol is attached to the handle of a child device after the child handle is created by the bus driver. The service in this protocol can return a bus specific override driver to `ConnectController()`.

`ConnectController()` must call this service until all of the bus specific override drivers have been retrieved. `ConnectController()` uses this information along with the EFI Platform Driver Override Protocol and all of the EFI Driver Binding protocol instances to select the best drivers for a controller. Since a controller can be managed by more than one driver, this protocol can return more than one bus specific override driver.

**EFI_BUS_SPECIFIC_DRIVER_OVERRIDE_PROTOCOL.GetDriver()**

Summary

Uses a bus specific algorithm to retrieve a driver image handle for a controller.

Prototype

```c
typedef
  EFI_STATUS
  (EFIAPI *EFI_BUS_SPECIFIC_DRIVER_OVERRIDE_GET_DRIVER) ( 
    IN   EFI_BUS_SPECIFIC_DRIVER_OVERRIDE_PROTOCOL *This,
    IN OUT EFI_HANDLE *DriverImageHandle
  );
```

Parameters

- **This**
  A pointer to the `EFI_BUS_SPECIFIC_DRIVER_OVERRIDE_PROTOCOL` instance.

- **DriverImageHandle**
  On input, a pointer to the previous driver image handle returned by `GetDriver()`. On output, a pointer to the next driver image handle. Passing in a **NULL**, will return the first driver image handle.
Description
This function is used to retrieve a driver image handle that is selected in a bus specific manner. The first
driver image handle is retrieved by passing in a `DriverImageHandle` value of `NULL`. This will cause the first
driver image handle to be returned in `DriverImageHandle`. On each successive call, the previous value of
`DriverImageHandle` must be passed in. If a call to this function returns a valid driver image handle, then
`EFI_SUCCESS` is returned. This process is repeated until `EFI_NOT_FOUND` is returned. If a
`DriverImageHandle` is passed in that was not returned on a prior call to this function, then
`EFI_INVALID_PARAMETER` is returned. The first driver image handle has the highest precedence, and
the last driver image handle has the lowest precedence. This ordered list of driver image handles is used
by the boot service `EFI_BOOT_SERVICES.ConnectController()` to search for the best driver for a
controller.

Status Codes Returned

<table>
<thead>
<tr>
<th>Status Code</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>EFI_SUCCESS</td>
<td>A bus specific override driver is returned in <code>DriverImageHandle</code>.</td>
</tr>
<tr>
<td>EFI_NOT_FOUND</td>
<td>The end of the list of override drivers was reached. A bus specific override driver is not returned in <code>DriverImageHandle</code>.</td>
</tr>
<tr>
<td>EFI_INVALID_PARAMETER</td>
<td><code>DriverImageHandle</code> is not a handle that was returned on a previous call to <code>GetDriver()</code>.</td>
</tr>
</tbody>
</table>

11.4 EFI Driver Diagnostics Protocol

This section provides a detailed description of the `EFI_DRIVER_DIAGNOSTICS2_PROTOCOL`. This is a
protocol that allows a UEFI driver to perform diagnostics on a controller that the driver is managing.

**EFI_DRIVER_DIAGNOSTICS2_PROTOCOL**

Summary
Used to perform diagnostics on a controller that a UEFI driver is managing.

GUID

```c
#define EFI_DRIVER_DIAGNOSTICS_PROTOCOL_GUID \ 
  {0x4d330321,0x025f,0x4aac,\ 
  {0x90,0xd8,0x5e,0xd9,0x00,0x17,0x3b,0x63}}
```

Protocol Interface Structure

```c
typedef struct _EFI_DRIVER_DIAGNOSTICS2_PROTOCOL {
  EFI_DRIVER_DIAGNOSTICS2_RUN_DIAGNOSTICS RunDiagnostics;
  CHAR8 *SupportedLanguages;
} EFI_DRIVER_DIAGNOSTICS2_PROTOCOL;
```

Parameters

- **RunDiagnostics** Runs diagnostics on a controller. See the `RunDiagnostics()` function description.
- **SupportedLanguages** A Null-terminated ASCII string that contains one or more supported language codes. This is the list of language codes that this protocol supports. The number of languages supported by a driver is up to the
driver writer. SupportedLanguages is specified in RFC 4646 format. See Appendix M for the format of language codes and language code arrays.

Description
The **EFI_DRIVER_DIAGNOSTICS2_PROTOCOL** is used by a platform management utility to allow the user to run driver specific diagnostics on a controller. This protocol is optionally attached to the image handle of driver in the driver's entry point. The platform management utility can collect all the **EFI_DRIVER_DIAGNOSTICS2_PROTOCOL** instances present in the system, and present the user with a menu of the controllers that have diagnostic capabilities. This platform management utility is invoked through a platform component such as the EFI Boot Manager.

**EFI_DRIVER_DIAGNOSTICS2_PROTOCOL.RunDiagnostics()**

Summary
Runs diagnostics on a controller.

Prototype

```c
typedef EFI_STATUS (EFIAPI *EFI_DRIVER_DIAGNOSTICS2_RUN_DIAGNOSTICS) (
    IN EFI_DRIVER_DIAGNOSTICS2_PROTOCOL *This,
    IN EFI_HANDLE ControllerHandle, 
    IN EFI_HANDLE ChildHandle OPTIONAL, 
    IN EFI_DRIVER_DIAGNOSTIC_TYPE DiagnosticType, 
    IN CHAR8 *Language, 
    OUT EFI_GUID **ErrorType, 
    OUT UINTN *BufferSize, 
    OUT CHAR16 **Buffer
);
```

Parameters

- **This**: A pointer to the **EFI_DRIVER_DIAGNOSTICS2_PROTOCOL** instance.
- **ControllerHandle**: The handle of the controller to run diagnostics on.
- **ChildHandle**: The handle of the child controller to run diagnostics on. This is an optional parameter that may be NULL. It will be NULL for device drivers. It will also be NULL for a bus drivers that attempt to run diagnostics on the bus controller. It will not be NULL for a bus driver that attempts to run diagnostics on one of its child controllers.
- **DiagnosticType**: Indicates type of diagnostics to perform on the controller specified by **ControllerHandle** and **ChildHandle**. See “Related Definitions” for the list of supported types.
- **Language**: A pointer to a Null-terminated ASCII string array indicating the language. This is the language in which the optional error message should be returned in Buffer, and it must match one of the languages specified in SupportedLanguages. The number of languages specified in SupportedLanguages is specified in RFC 4646 format. See Appendix M for the format of language codes and language code arrays.
supported by a driver is up to the driver writer. Language is specified in RFC 4646 language code format. See Appendix M for the format of language codes.

Callers of interfaces that require RFC 4646 language codes to retrieve a Unicode string must use the RFC 4647 algorithm to lookup the Unicode string with the closest matching RFC 4646 language code.

**ErrorType**

A GUID that defines the format of the data returned in Buffer.

**BufferSize**

The size, in bytes, of the data returned in Buffer.

**Buffer**

A buffer that contains a Null-terminated string plus some additional data whose format is defined by ErrorType. Buffer is allocated by this function with EFI_BOOT_SERVICES.AllocatePool(), and it is the caller’s responsibility to free it with a call to EFI_BOOT_SERVICES.FreePool().

**Description**

This function runs diagnostics on the controller specified by ControllerHandle and ChildHandle. DiagnosticType specifies the type of diagnostics to perform on the controller specified by ControllerHandle and ChildHandle. If the driver specified by This does not support the language specified by Language, then EFI_UNSUPPORTED is returned. If the controller specified by ControllerHandle and ChildHandle is not supported by the driver specified by This, then EFI_UNSUPPORTED is returned. If the diagnostics type specified by DiagnosticType is not supported by this driver, then EFI_UNSUPPORTED is returned. If there are not enough resources available to complete the diagnostic, then EFI_OUT_OF_RESOURCES is returned. If the controller specified by ControllerHandle and ChildHandle passes the diagnostic, then EFI_SUCCESS is returned. Otherwise, EFI_DEVICE_ERROR is returned.

If the language specified by Language is supported by this driver, then status information is returned in ErrorType, BufferSize, and Buffer. Buffer contains a Null-terminated string followed by additional data whose format is defined by ErrorType. BufferSize is the size of Buffer is bytes, and it is the caller’s responsibility to call FreePool() on Buffer when the caller is done with the return data. If there are not enough resources available to return the status information, then EFI_OUT_OF_RESOURCES is returned.
Related Definitions

//*******************************************************
// EFI_DRIVER_DIAGNOSTIC_TYPE
//*****************************************************************************
typedef enum {
  EfiDriverDiagnosticTypeStandard        = 0,
  EfiDriverDiagnosticTypeExtended        = 1,
  EfiDriverDiagnosticTypeManufacturing   = 2,
  EfiDriverDiagnosticTypeCancel          = 3,
  EfiDriverDiagnosticTypeMaximum
} EFI_DRIVER_DIAGNOSTIC_TYPE;

EfiDriverDiagnosticTypeStandard
  Performs standard diagnostics on the controller. This diagnostic type is required to
  be supported by all implementations of this protocol.

EfiDriverDiagnosticTypeExtended
  This is an optional diagnostic type that performs diagnostics on the controller that
  may take an extended amount of time to execute.

EfiDriverDiagnosticTypeManufacturing
  This is an optional diagnostic type that performs diagnostics on the controller that
  are suitable for a manufacturing and test environment.

EfiDriverDiagnosticTypeCancel
  This is an optional diagnostic type that would only be used in the situation where an
  EFI_NOT_READY had been returned by a previous call to RunDiagnostics() and
  there is a desire to cancel the current running diagnostics operation.
11.5 EFI Component Name Protocol

This section provides a detailed description of the **EFI_COMPONENT_NAME2_PROTOCOL**. This is a protocol that allows an driver to provide a user readable name of a UEFI Driver, and a user readable name for each of the controllers that the driver is managing. This protocol is used by platform management utilities that wish to display names of components. These names may include the names of expansion slots, external connectors, embedded devices, and add-in devices.

**EFI_COMPONENT_NAME2_PROTOCOL**

**Summary**

Used to retrieve user readable names of drivers and controllers managed by UEFI Drivers.

---

**Status Codes Returned**

<table>
<thead>
<tr>
<th>Status Code</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>EFI_SUCCESS</td>
<td>The controller specified by <code>ControllerHandle</code> and <code>ChildHandle</code> passed the diagnostic.</td>
</tr>
<tr>
<td>EFI_ACCESS_DENIED</td>
<td>The request for initiating diagnostics was unable to be completed due to some underlying hardware or software state.</td>
</tr>
<tr>
<td>EFI_INVALID_PARAMETER</td>
<td><code>ControllerHandle</code> is <strong>NULL</strong>.</td>
</tr>
<tr>
<td>EFI_INVALID_PARAMETER</td>
<td>The driver specified by <code>This</code> is not a device driver, and <code>ChildHandle</code> is not <strong>NULL</strong>.</td>
</tr>
<tr>
<td>EFI_INVALID_PARAMETER</td>
<td><code>Language</code> is <strong>NULL</strong>.</td>
</tr>
<tr>
<td>EFI_INVALID_PARAMETER</td>
<td><code>ErrorType</code> is <strong>NULL</strong>.</td>
</tr>
<tr>
<td>EFI_INVALID_PARAMETER</td>
<td><code>BufferSize</code> is <strong>NULL</strong>.</td>
</tr>
<tr>
<td>EFI_INVALID_PARAMETER</td>
<td><code>Buffer</code> is <strong>NULL</strong>.</td>
</tr>
<tr>
<td>EFI_UNSUPPORTED</td>
<td>The driver specified by <code>This</code> does not support running diagnostics for the controller specified by <code>ControllerHandle</code> and <code>ChildHandle</code>.</td>
</tr>
<tr>
<td>EFI_UNSUPPORTED</td>
<td>The driver specified by <code>This</code> does not support the type of diagnostic specified by <code>DiagnosticType</code>.</td>
</tr>
<tr>
<td>EFI_UNSUPPORTED</td>
<td>The driver specified by <code>This</code> does not support the language specified by <code>Language</code>.</td>
</tr>
<tr>
<td>EFI_OUT_OF_RESOURCES</td>
<td>There are not enough resources available to complete the diagnostics.</td>
</tr>
<tr>
<td>EFI_OUT_OF_RESOURCES</td>
<td>There are not enough resources available to return the status information in <code>ErrorType</code>, <code>BufferSize</code>, and <code>Buffer</code>.</td>
</tr>
<tr>
<td>EFI_DEVICE_ERROR</td>
<td>The controller specified by <code>ControllerHandle</code> and <code>ChildHandle</code> did not pass the diagnostic.</td>
</tr>
<tr>
<td>EFI_NOT_READY</td>
<td>The diagnostic operation was started, but not yet completed.</td>
</tr>
</tbody>
</table>
GUID

#define EFI_COMPONENT_NAME2_PROTOCOL_GUID \ 
{0x6a7a5cfe, 0xe8d9, 0x4f70,\ 
 {0xba, 0xda, 0x75, 0xab, 0x30,0x25, 0xce, 0x14}}

Protocol Interface Structure

typedef struct _EFI_COMPONENT_NAME2_PROTOCOL {
  EFI_COMPONENT_NAME_GET_DRIVER_NAME GetDriverName;
  EFI_COMPONENT_NAME_GET_CONTROLLER_NAME GetControllerName;
  CHAR8 SupportedLanguages;
} EFI_COMPONENT_NAME2_PROTOCOL;

Parameters

GetDriverName Retrieves a string that is the user readable name of the driver. See the GetDriverName() function description.

GetControllerName Retrieves a string that is the user readable name of a controller that is being managed by a driver. See the GetControllerName() function description.

SupportedLanguages A Null-terminated ASCII string array that contains one or more supported language codes. This is the list of language codes that this protocol supports. The number of languages supported by a driver is up to the driver writer. SupportedLanguages is specified in RFC 4646 format. See Appendix M for the format of language codes and language code arrays.

Description

The EFI_COMPONENT_NAME2_PROTOCOL is used retrieve a driver's user readable name and the names of all the controllers that a driver is managing from the driver's point of view. Each of these names is returned as a Null-terminated string. The caller is required to specify the language in which the string is returned, and this language must be present in the list of languages that this protocol supports specified by SupportedLanguages.

EFI_COMPONENT_NAME2_PROTOCOL.GetDriverName()

Summary

Retrieves a string that is the user readable name of the driver.
Prototype

```c
typedef EFI_STATUS
  (EFIAPI *EFI_COMPONENT_NAME_GET_DRIVER_NAME) (
  IN EFI_COMPONENT_NAME2_PROTOCOL *This,
  IN CHAR8 *Language,
  OUT CHAR16 **DriverName
  );
```

Parameters

- **This**
  A pointer to the EFI_COMPONENT_NAME2_PROTOCOL instance.

- **Language**
  A pointer to a Null-terminated ASCII string array indicating the language. This is the language of the driver name that the caller is requesting, and it must match one of the languages specified in SupportedLanguages. The number of languages supported by a driver is up to the driver writer. Language is specified in RFC 4646 language code format. See Appendix M for the format of language codes.

  Callers of interfaces that require RFC 4646 language codes to retrieve a Unicode string must use the RFC 4647 algorithm to lookup the Unicode string with the closest matching RFC 4646 language code.

- **DriverName**
  A pointer to the string to return. This string is the name of the driver specified by **This** in the language specified by **Language**.

Description

This function retrieves the user readable name of a driver in the form of a string. If the driver specified by **This** has a user readable name in the language specified by **Language**, then a pointer to the driver name is returned in **DriverName**, and EFI_SUCCESS is returned. If the driver specified by **This** does not support the language specified by **Language**, then EFI_UNSUPPORTED is returned.

Status Codes Returned

<table>
<thead>
<tr>
<th>Status Code</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>EFI_SUCCESS</td>
<td>The string for the user readable name in the language specified by <strong>Language</strong> for the driver specified by <strong>This</strong> was returned in <strong>DriverName</strong>.</td>
</tr>
<tr>
<td>EFI_INVALID_PARAMETER</td>
<td><strong>Language</strong> is NULL.</td>
</tr>
<tr>
<td>EFI_INVALID_PARAMETER</td>
<td><strong>DriverName</strong> is NULL.</td>
</tr>
<tr>
<td>EFI_UNSUPPORTED</td>
<td>The driver specified by <strong>This</strong> does not support the language specified by <strong>Language</strong>.</td>
</tr>
</tbody>
</table>

**EFI_COMPONENT_NAME2_PROTOCOL.GetControllerName()**

Summary

Retrieves a string that is the user readable name of the controller that is being managed by a driver.
Prototype

```c
typedef
EFI_STATUS
(EFIAPI *EFI_COMPONENT_NAME_GET_CONTROLLER_NAME) (
    IN EFI_COMPONENT_NAME2_PROTOCOL *This,
    IN EFI_HANDLE ControllerHandle,
    IN EFI_HANDLE ChildHandle OPTIONAL,
    IN CHAR8 *Language,
    OUT CHAR16 **ControllerName
);
```

Parameters

- **This**: A pointer to the EFI_COMPONENT_NAME2_PROTOCOL instance.
- **ControllerHandle**: The handle of a controller that the driver specified by `This` is managing. This handle specifies the controller whose name is to be returned.
- **ChildHandle**: The handle of the child controller to retrieve the name of. This is an optional parameter that may be `NULL`. It will be `NULL` for device drivers. It will also be `NULL` for bus drivers that attempt to retrieve the name of the bus controller. It will not be `NULL` for a bus driver that attempts to retrieve the name of a child controller.
- **Language**: A pointer to a Null-terminated ASCII string array indicating the language. This is the language of the controller name that the caller is requesting, and it must match one of the languages specified in SupportedLanguages. The number of languages supported by a driver is up to the driver writer. `Language` is specified in RFC 4646 language code format. See Appendix M for the format of language codes.

Callers of interfaces that require RFC 4646 language codes to retrieve a Unicode string must use the RFC 4647 algorithm to lookup the Unicode string with the closest matching RFC 4646 language code.

- **ControllerName**: A pointer to the string to return. This string is the name of the controller specified by `ControllerHandle` and `ChildHandle` in the language specified by `Language` from the point of view of the driver specified by `This`.

Description

This function retrieves the user readable name of the controller specified by `ControllerHandle` and `ChildHandle` in the form of a string. If the driver specified by `This` has a user readable name in the language specified by `Language`, then a pointer to the controller name is returned in `ControllerName`, and **EFI_SUCCESS** is returned.

If the driver specified by `This` is not currently managing the controller specified by `ControllerHandle` and `ChildHandle`, then **EFI_UNSUPPORTED** is returned.

If the driver specified by `This` does not support the language specified by `Language`, then **EFI_UNSUPPORTED** is returned.
## Status Codes Returned

<table>
<thead>
<tr>
<th>Code</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>EFI_SUCCESS</td>
<td>The string for the user readable name specified by This, ControllerHandle, ChildHandle, and Language was returned in ControllerName.</td>
</tr>
<tr>
<td>EFI_INVALID_PARAMETER</td>
<td>ControllerHandle is <strong>NULL</strong>.</td>
</tr>
<tr>
<td>EFI_INVALID_PARAMETER</td>
<td>The driver specified by This is not a device driver, and ChildHandle is <strong>NULL</strong>.</td>
</tr>
<tr>
<td>EFI_INVALID_PARAMETER</td>
<td>Language is <strong>NULL</strong>.</td>
</tr>
<tr>
<td>EFI_INVALID_PARAMETER</td>
<td>ControllerName is <strong>NULL</strong>.</td>
</tr>
<tr>
<td>EFI_UNSUPPORTED</td>
<td>The driver specified by This is a device driver and ChildHandle is not <strong>NULL</strong>.</td>
</tr>
<tr>
<td>EFI_UNSUPPORTED</td>
<td>The driver specified by This is not currently managing the controller specified by ControllerHandle and ChildHandle.</td>
</tr>
<tr>
<td>EFI_UNSUPPORTED</td>
<td>The driver specified by This does not support the language specified by Language.</td>
</tr>
</tbody>
</table>

### 11.6 EFI Service Binding Protocol

This section provides a detailed description of the **EFI_SERVICE_BINDING_PROTOCOL**. This protocol may be produced only by drivers that follow the UEFI Driver Model. Use this protocol with the **EFI_DRIVER_BINDING_PROTOCOL** to produce a set of protocols related to a device. The **EFI_DRIVER_BINDING_PROTOCOL** supports simple layering of protocols on a device, but it does not support more complex relationships such as trees or graphs. The **EFI_SERVICE_BINDING_PROTOCOL** provides a member function to create a child handle with a new protocol installed on it, and another member function to destroy a previously created child handle. These member functions apply equally to all drivers.

**EFI_SERVICE_BINDING_PROTOCOL**

**Summary**

Provides services that are required to create and destroy child handles that support a given set of protocols.

**GUID**

This protocol does not have its own GUID. Instead, drivers for other protocols will define a GUID that shares the same protocol interface as the ** EFI_SERVICE_BINDING_PROTOCOL**. The protocols defined in this document that have this property include for example the following:

- ** EFI_MANAGED_NETWORK_SERVICE_BINDING_PROTOCOL**
- ** EFI_ARP_SERVICE_BINDING_PROTOCOL**
- ** EFI_EAP_SERVICE_BINDING_PROTOCOL**
- ** EFI_IP4_SERVICE_BINDING_PROTOCOL**
- ** EFI_TCP4_SERVICE_BINDING_PROTOCOL**
- ** EFI_UDP4_SERVICE_BINDING_PROTOCOL**
• EFI_MTFTP4_SERVICE_BINDING_PROTOCOL
• EFI_DHCP4_SERVICE_BINDING_PROTOCOL
• EFI_REST_EX_SERVICE_BINDING_PROTOCOL

Protocol Interface Structure

typedef struct _EFI_SERVICE_BINDING_PROTOCOL {
    EFI_SERVICE_BINDING_CREATE_CHILD CreateChild;
    EFI_SERVICE_BINDING_DESTROY_CHILD DestroyChild;
} EFI_SERVICE_BINDING_PROTOCOL;

Parameters

CreateChild  Creates a child handle and installs a protocol. See the CreateChild() function description.

DestroyChild  Destroys a child handle with a protocol installed on it. See the DestroyChild() function description.

Description

The EFI_SERVICE_BINDING_PROTOCOL provides member functions to create and destroy child handles. A driver is responsible for adding protocols to the child handle in CreateChild() and removing protocols in DestroyChild(). It is also required that the CreateChild() function opens the parent protocol BY_CHILD_CONTROLLER to establish the parent-child relationship, and closes the protocol in DestroyChild(). The pseudo code for CreateChild() and DestroyChild() is provided to specify the required behavior, not to specify the required implementation. Each consumer of a software protocol is responsible for calling CreateChild() when it requires the protocol and calling DestroyChild() when it is finished with that protocol.

EFI_SERVICE_BINDING_PROTOCOL.CreateChild()

Summary

Creates a child handle and installs a protocol.

Prototype

typedef
EFI_STATUS
(EIFIAPI *EFI_SERVICE_BINDING_CREATE_CHILD) (
    IN EFI_SERVICE_BINDING_PROTOCOL *This,
    IN OUT EFI_HANDLE *ChildHandle
);

Parameters

This  Pointer to the EFI_SERVICE_BINDING_PROTOCOL instance.

ChildHandle  Pointer to the handle of the child to create. If it is a pointer to NULL, then a new handle is created. If it is a pointer to an existing UEFI handle, then the protocol is added to the existing UEFI handle.
Description

The `CreateChild()` function installs a protocol on `ChildHandle`. If `ChildHandle` is a pointer to `NULL`, then a new handle is created and returned in `ChildHandle`. If `ChildHandle` is not a pointer to `NULL`, then the protocol installs on the existing `ChildHandle`.

Status Codes Returned

<table>
<thead>
<tr>
<th>Status Code</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>EFI_SUCCESS</td>
<td>The protocol was added to <code>ChildHandle</code>.</td>
</tr>
<tr>
<td>EFI_INVALID_PARAMETER</td>
<td><code>ChildHandle</code> is <code>NULL</code>.</td>
</tr>
<tr>
<td>EFI_OUT_OF_RESOURCES</td>
<td>There are not enough resources available to create the child.</td>
</tr>
<tr>
<td>Other</td>
<td>The child handle was not created.</td>
</tr>
</tbody>
</table>

Examples

The following example shows how a consumer of the EFI ARP Protocol would use the `CreateChild()` function of the `EFI_SERVICE_BINDING_PROTOCOL` to create a child handle with the EFI ARP Protocol installed on that handle.

```c
EFI_HANDLE                       ControllerHandle;
EFI_HANDLE                       DriverBindingHandle;
EFI_HANDLE                       ChildHandle;
EFI_ARP_SERVICE_BINDING_PROTOCOL *ArpSb;
EFI_ARP_PROTOCOL                 *Arp;

// // Get the ArpServiceBinding Protocol
//
Status = gBS->OpenProtocol (    ControllerHandle,    &gEfiArpServiceBindingProtocolGuid,    (VOID **)&ArpSb,    DriverBindingHandle,    ControllerHandle,    EFI_OPEN_PROTOCOL_GET_PROTOCOL    );
if (EFI_ERROR (Status)) {
    return Status;
}
// // Initialize a ChildHandle
//
ChildHandle = NULL;
// // Create a ChildHandle with the Arp Protocol
//
Status = ArpSb->CreateChild (ArpSb, &ChildHandle);
if (EFI_ERROR (Status)) {
    goto ErrorExit;
}
```
// Retrieve the Arp Protocol from ChildHandle
//
// STATUS = gBS->OpenProtocol (ChildHandle,
// &gEfiArpProtocolGuid,
// (VOID **)&Arp,
// DriverBindingHandle,
// ControllerHandle,
// EFI_OPEN_PROTOCOL_BY_DRIVER);
//
// if (EFI_ERROR (Status)) {
//   goto ErrorExit;
// }

**Pseudo Code**

The following is the general algorithm for implementing the `CreateChild()` function:

1. Allocate and initialize any data structures that are required to produce the requested protocol on a child handle. If the allocation fails, then return `EFI_OUT_OF_RESOURCES`.
2. Install the requested protocol onto `ChildHandle`. If `ChildHandle` is a pointer to `NULL`, then the requested protocol installs onto a new handle.
3. Open the parent protocol `BY_CHILD_CONTROLLER` to establish the parent-child relationship. If the parent protocol cannot be opened, then destroy the `ChildHandle` created in step 2, free the data structures allocated in step 1, and return an error.
4. Increment the number of children created by `CreateChild()`.
5. Return `EFI_SUCCESS`.

Listed below is sample code of the `CreateChild()` function of the EFI ARP Protocol driver. This driver looks up its private context data structure from the instance of the `EFI_SERVICE_BINDING_PROTOCOL` produced on the handle for the network controller. After retrieving the private context data structure, the driver can use its contents to build the private context data structure for the child being created. The EFI ARP Protocol driver then installs the `EFI_ARP_PROTOCOL` onto `ChildHandle`.

```c
EFI_STATUS
EFIAPI
ArpServiceBindingCreateChild (IN EFI_SERVICE_BINDING_PROTOCOL *This,
    IN EFI_HANDLE *ChildHandle)
{
    EFI_STATUS Status;
    ARP_PRIVATE_DATA *Private;
    ARP_PRIVATE_DATA *PrivateChild;
    
    // Retrieve the Private Context Data Structure
    Private = ARP_PRIVATE_DATA_FROM_SERVICE_BINDING_THIS (This);
```
// Create a new child
//
PrivateChild = EfiLibAllocatePool (sizeof (ARP_PRIVATE_DATA));
if (PrivateChild == NULL) {
    return EFI_OUT_OF_RESOURCES;
}

// Copy Private Context Data Structure
//
//gBS->CopyMem (PrivateChild, Private, sizeof (ARP_PRIVATE_DATA));

// Install Arp onto ChildHandle
//
Status = gBS->InstallMultipleProtocolInterfaces (ChildHandle,
    &gEfiArpProtocolGuid, &PrivateChild->Arp, NULL);
if (EFI_ERROR (Status)) {
    gBS->FreePool (PrivateChild);
    return Status;
}

Status = gBS->OpenProtocol (Private->ChildHandle,
    &gEfiManagedNetworkProtocolGuid,
    (VOID **)&PrivateChild->ManagedNetwork,
    gArpDriverBinding.DriverBindingHandle,
    *ChildHandle,
    EFI_OPEN_PROTOCOL_BY_CHILD_CONTROLLER);
if (EFI_ERROR (Status)) {
    ArpSB->DestroyChild (This, ChildHandle);
    return Status;
}

// Increase number of children created
//
Private->NumberCreated++;

return EFI_SUCCESS;
EFI_SERVICE_BINDING_PROTOCOL.DestroyChild()

Summary
Destroys a child handle with a protocol installed on it.

Prototype

```c
typedef
EFI_STATUS
(EIFIAPI *EFI_SERVICE_BINDING_DESTROY_CHILD)(
  IN EFI_SERVICE_BINDING_PROTOCOL  *This,
  IN EFI_HANDLE                   ChildHandle
);
```

Parameters

- **This**: Pointer to the EFI_SERVICE_BINDING_PROTOCOL instance.
- **ChildHandle**: Handle of the child to destroy.

Description
The **DestroyChild()** function does the opposite of **CreateChild()**. It removes a protocol that was installed by **CreateChild()** from **ChildHandle**. If the removed protocol is the last protocol on **ChildHandle**, then **ChildHandle** is destroyed.

Status Codes Returned

<table>
<thead>
<tr>
<th>Status Code</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>EFI_SUCCESS</td>
<td>The protocol was removed from <strong>ChildHandle</strong>.</td>
</tr>
<tr>
<td>EFI_UNSUPPORTED</td>
<td><strong>ChildHandle</strong> does not support the protocol that is being removed.</td>
</tr>
<tr>
<td>EFI_INVALID_PARAMETER</td>
<td><strong>ChildHandle</strong> is not a valid UEFI handle.</td>
</tr>
<tr>
<td>EFI_ACCESS_DENIED</td>
<td>The protocol could not be removed from the <strong>ChildHandle</strong> because its services are being used.</td>
</tr>
<tr>
<td>Other</td>
<td>The child handle was not destroyed.</td>
</tr>
</tbody>
</table>

Examples

The following example shows how a consumer of the EFI ARP Protocol would use the **DestroyChild()** function of the EFI_SERVICE_BINDING_PROTOCOL to destroy a child handle with the EFI ARP Protocol installed on that handle.

```c
EFI_HANDLE ControllerHandle;
EFI_HANDLE DriverBindingHandle;
EFI_HANDLE ChildHandle;
EFI_ARP_SERVICE_BINDING_PROTOCOL  *Arp;

//
// Get the Arp Service Binding Protocol
//
Status = gBS->OpenProtocol (ControllerHandle,
```
&gEfiArpServiceBindingProtocolGuid,
  (VOID **)ArpSb,
  DriverBindingHandle,
  ControllerHandle,
  EFI_OPEN_PROTOCOL_GET_PROTOCOL

if (EFI_ERROR (Status)) {
  return Status;
}

// Destroy the ChildHandle with the Arp Protocol
//
// Status = ArpSb->DestroyChild (ArpSb, ChildHandle);
// if (EFI_ERROR (Status)) {
//  return Status;
//}

Pseudo Code

The following is the general algorithm for implementing the DestroyChild() function:

1. Retrieve the protocol from ChildHandle. If this retrieval fails, then return EFI_SUCCESS because the child has already been destroyed.
2. If this call is a recursive call to destroy the same child, then return EFI_SUCCESS.
3. Close the parent protocol with CloseProtocol().
4. Set a flag to detect a recursive call to destroy the same child.
5. Remove the protocol from ChildHandle. If this removal fails, then reopen the parent protocol and clear the flag to detect a recursive call to destroy the same child.
6. Free any data structures that allocated in CreateChild().
7. Decrement the number of children that created with CreateChild().
8. Return EFI_SUCCESS.

Listed below is sample code of the DestroyChild() function of the EFI ARP Protocol driver. This driver looks up its private context data structure from the instance of the EFI_SERVICE_BINDING_PROTOCOL produced on the handle for the network controller. The driver attempts to retrieve the EFI_ARP_PROTOCOL from ChildHandle. If that fails, then EFI_SUCCESS is returned. The EFI_ARP_PROTOCOL is then used to retrieve the private context data structure for the child. The private context data stores the flag that detects if DestroyChild() is being called recursively. If a recursion is detected, then EFI_SUCCESS is returned. Otherwise, the EFI_ARP_PROTOCOL is removed from ChildHandle, the number of children are decremented, and EFI_SUCCESS is returned.

EFI_STATUS
EFIAPI
ArpServiceBindingDestroyChild (  
  IN EFI_SERVICE_BINDING_PROTOCOL *This,
  IN EFI_HANDLE ChildHandle
 )
{
    EFI_STATUS Status;
    EFI_ARP_PROTOCOL *Arp;
    ARP_PRIVATE_DATA *Private;
    ARP_PRIVATE_DATA *PrivateChild;

    // Retrieve the Private Context Data Structure
    //
    Private = ARP_PRIVATE_DATA_FROM_SERVICE_BINDING_THIS(This);

    // Retrieve Arp Protocol from ChildHandle
    //
    Status = gBS->OpenProtocol(
        ChildHandle,
        &gEfiArpProtocolGuid,
        (VOID **)&Arp,
        gArpDriverBinding.DriverBindingHandle,
        ChildHandle,
        EFI_OPEN_PROTOCOL_GET_PROTOCOL);
    if (EFI_ERROR(Status)) {
        return EFI_SUCCESS;
    }

    // Retrieve Private Context Data Structure
    //
    PrivateChild = ARP_PRIVATE_DATA_FROM_ARP_THIS(Arp);
    if (PrivateChild->Destroy) {
        return EFI_SUCCESS;
    }

    // Close the ManagedNetwork Protocol
    //
    gBS->CloseProtocol(
        Private->ChildHandle,
        &gEfiManagedNetworkProtocolGuid,
gArpDriverBinding.DriverBindingHandle,
  ChildHandle
);

PrivateChild->Destroy = TRUE;

//
// Uninstall Arp from ChildHandle
//
Status = gBS->UninstallMultipleProtocolInterfaces (  
  ChildHandle,
  &gEfiArpProtocolGuid, &PrivateChild->Arp,
  NULL
);
if (EFI_ERROR (Status)) {
  //
  // Uninstall failed, so reopen the parent Arp Protocol and
  // return an error
  //
  PrivateChild->Destroy = FALSE;
  gBS->OpenProtocol (  
    Private->ChildHandle,
    &gEfiManagedNetworkProtocolGuid,
    (VOID **)&PrivateChild->ManagedNetwork,
    gArpDriverBinding.DriverBindingHandle,
    ChildHandle,
    EFI_OPEN_PROTOCOL_BY_CHILD_CONTROLLER
  );
  return Status;
}

//
// Free Private Context Data Structure
//
gBS->FreePool (PrivateChild);

//
// Decrease number of children created
//
Private->NumberCreated--;

return EFI_SUCCESS;
11.7 EFI Platform to Driver Configuration Protocol

This section provides a detailed description of the EFI_PLATFORM_TO_DRIVER_CONFIGURATION_PROTOCOL. This is a protocol that is optionally produced by the platform and optionally consumed by a UEFI Driver in its Start() function. This protocol allows the driver to receive configuration information as part of being started.

**EFI_PLATFORM_TO_DRIVER_CONFIGURATION_PROTOCOL**

**Summary**

Used to retrieve configuration information for a device that a UEFI driver is about to start.

**GUID**

```
#define EFI_PLATFORM_TO_DRIVER_CONFIGURATION_PROTOCOL_GUID \
{ 0x642cd590, 0x8059, 0x4c0a,\n  { 0xa9, 0x58, 0xc5, 0xec, 0x07, 0xd2, 0x3c, 0x4b } }
```

**Protocol Interface Structure**

```
typedef struct _EFI_PLATFORM_TO_DRIVER_CONFIGURATION_PROTOCOL {
  EFI_PLATFORM_TO_DRIVER_CONFIGURATION_QUERY Query;
  EFIPLATFORM_TO_DRIVER_CONFIGURATION_RESPONSE Response;
} EFI_PLATFORM_TO_DRIVER_CONFIGURATION_PROTOCOL;
```

**Parameters**

- **Query**
  Called by the UEFI Driver Start() function to get configuration information from the platform.

- **Response**
  Called by the UEFI Driver Start() function to let the platform know how UEFI driver processed the data return from Query.

**Description**

The EFI_PLATFORM_TO_DRIVER_CONFIGURATION_PROTOCOL is used by the UEFI driver to query the platform for configuration information. The UEFI driver calls Query() multiple times to get configuration information from the platform. For every call to Query() there must be a matching call to Response() so the UEFI driver can inform the platform how it used the information passed in from Query().

It’s legal for a UEFI driver to use Response() to inform the platform it does not understand the data returned via Query() and thus no action was taken.

**EFI_PLATFORM_TO_DRIVER_CONFIGURATION_PROTOCOL.Query()**

**Summary**

Allows the UEFI driver to query the platform for configuration information needed to complete the drivers Start() operation.
Prototype

```c
typedef EFI_STATUS
  (EFIAPI *EFI_PLATFORM_TO_DRIVER_CONFIGURATION_QUERY) ( 
    IN EFI_PLATFORM_TO_DRIVER_CONFIGURATION_PROTOCOL *This, 
    IN EFI_HANDLE ControllerHandle, 
    IN EFI_HANDLE ChildHandle OPTIONAL, 
    IN UINTN *Instance, 
    OUT EFI_GUID **ParameterTypeGuid, 
    OUT VOID **ParameterBlock, 
    OUT UINTN *ParameterBlockSize 
  );
```

Parameters

- **This**: A pointer to the `EFI_PLATFORM_TO_DRIVER_CONFIGURATION_PROTOCOL` instance.
- **ControllerHandle**: The handle the platform will return configuration information about.
- **ChildHandle**: The handle of the child controller to return information on. This is an optional parameter that may be `NULL`. It will be `NULL` for device drivers, and for bus drivers that attempt to get options for the bus controller. It will not be `NULL` for a bus driver that attempts to get options for one of its child controllers.
- **Instance**: Pointer to the Instance value. Zero means return the first query data. The caller should increment this value by one each time to retrieve successive data.
- **ParameterTypeGuid**: An `EFI_GUID` that defines the contents of `ParameterBlock`. UEFI drivers must use the `ParameterTypeGuid` to determine how to parse the `ParameterBlock`. The caller should not attempt to free `ParameterTypeGuid`.
- **ParameterBlock**: The platform returns a pointer to the `ParameterBlock` structure which contains details about the configuration parameters specific to the `ParameterTypeGuid`. This structure is defined based on the protocol and may be different for different protocols. UEFI driver decodes this structure and its contents based on `ParameterTypeGuid`. `ParameterBlock` is allocated by the platform and the platform is responsible for freeing the `ParameterBlock` after `Response` is called.
- **ParameterBlockSize**: The platform returns the size of the `ParameterBlock` in bytes.

Description

The UEFI driver must call `Query` early in the `Start()` function before any time consuming operations are performed. If `ChildHandle` is `NULL` the driver is requesting information from the platform about the `ControllerHandle` that is being started. Information returned from `Query` may lead to the drivers `Start()` function failing.
If the UEFI driver is a bus driver and producing a `ChildHandle` the driver must call `Query` after the child handle has been created and an `EFI_DEVICE_PATH_PROTOCOL` has been placed on that handle, but before any time consuming operation is performed. If information return by `Query` may lead the driver to decide to not create the `ChildHandle`. The driver must then cleanup and remove the `ChildHandle` from the system.

The UEFI driver repeatedly calls `Query`, processes the information returned by the platform, and calls `Response` passing in the arguments returned from `Query`. The `Instance` value passed into `Response` must be the same value passed to the corresponding call to `Query`. The UEFI driver must continuously call `Query` and `Response` until `EFI_NOT_FOUND` is returned by `Query`.

If the UEFI driver does not recognize the `ParameterTypeGuid`, it calls `Response` with a `ConfigurationAction` of `EfiPlatformConfigurationActionUnsupportedGuid`. The UEFI driver must then continue calling `Query` and `Response` until `EFI_NOT_FOUND` is returned by `Query`. This gives the platform an opportunity to pass additional configuration settings using a different `ParameterTypeGuid` that may be supported by the driver.

An `Instance` value of zero means return the first `ParameterBlock` in the set of unprocessed parameter blocks. The driver should increment the `Instance` value by one for each successive call to `Query`.

**Status Codes Returned**

<table>
<thead>
<tr>
<th>Status Code</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>EFI_SUCCESS</td>
<td>The platform return parameter information for <code>ControllerHandle</code>.</td>
</tr>
<tr>
<td>EFI_NOT_FOUND</td>
<td>No more unread <code>Instance</code> exists.</td>
</tr>
<tr>
<td>EFI_INVALID_PARAMETER</td>
<td><code>ControllerHandle</code> is <code>NULL</code>.</td>
</tr>
<tr>
<td>EFI_INVALID_PARAMETER</td>
<td><code>Instance</code> is <code>NULL</code>.</td>
</tr>
<tr>
<td>EFI_DEVICE_ERROR</td>
<td>A device error occurred while attempting to return parameter block information for the controller specified by <code>ControllerHandle</code> and <code>ChildHandle</code>.</td>
</tr>
<tr>
<td>EFI_OUT_RESOURCES</td>
<td>There are not enough resources available to set the configuration options for the controller specified by <code>ControllerHandle</code> and <code>ChildHandle</code>.</td>
</tr>
</tbody>
</table>

**EFI_PLATFORM_TO_DRIVER_CONFIGURATION_PROTOCOL.Response()**

**Summary**

Tell the platform what actions where taken by the driver after processing the data returned from `Query`. 
Prototype

```c
typedef
    EFI_STATUS
    (EFIAPI *EFI_PLATFORM_TO_DRIVER_CONFIGURATION_RESPONSE) (
        IN EFI_PLATFORM_TO_DRIVER_CONFIGURATION_PROTOCOL *This,
        IN EFI_HANDLE ControllerHandle,
        IN EFI_HANDLE ChildHandle OPTIONAL,
        IN UINTN *Instance,
        IN EFI_GUID *ParameterTypeGuid,
        IN VOID *ParameterBlock,
        IN UINTN ParameterBlockSize,
        IN EFI_PLATFORM_CONFIGURATION_ACTION ConfigurationAction
    );
```

Parameters

- **This**: A pointer to the `EFI_PLATFORM_TO_DRIVER_CONFIGURATION_PROTOCOL` instance.
- **ControllerHandle**: The handle the driver is returning configuration information about.
- **ChildHandle**: The handle of the child controller to return information on. This is an optional parameter that may be `NULL`. It will be `NULL` for device drivers, and for bus drivers that attempt to get options for the bus controller. It will not be `NULL` for a bus driver that attempts to get options for one of its child controllers.
- **Instance**: Instance data passed to `Query()`.
- **ParameterTypeGuid**: `ParameterTypeGuid` returned from `Query`.
- **ParameterBlock**: `ParameterBlock` returned from `Query`.
- **ParameterBlockSize**: The `ParameterBlock` size returned from `Query`.
- **ConfigurationAction**: The driver tells the platform what action is required for `ParameterBlock` to take effect. See “Related Definitions” for a list of actions.

Description

The UEFI driver repeatedly calls `Query`, processes the information returned by the platform, and calls `Response` passing in the arguments returned from `Query`. The UEFI driver must continuously call `Query` until `EFI_NOT_FOUND` is returned. For every call to `Query` that returns `EFI_SUCCESS` a corresponding call to `Response` is required passing in the same `ControllerHandle`, `ChildHandle`, `Instance`, `ParameterTypeGuid`, `ParameterBlock`, and `ParameterBlockSize`. The UEFI driver may update values in `ParameterBlock` based on rules defined by `ParameterTypeGuid`.

The platform is responsible for freeing `ParameterBlock` and the UEFI driver must not try to free it.
typedef enum {
    EfiPlatformConfigurationActionNone          = 0,
    EfiPlatformConfigurationActionStopController = 1,
    EfiPlatformConfigurationActionRestartController = 2,
    EfiPlatformConfigurationActionRestartPlatform = 3,
    EfiPlatformConfigurationActionNvramFailed     = 4,
    EfiPlatformConfigurationActionUnsupportedGuid = 5,
    EfiPlatformConfigurationActionMaximum
} EFI_PLATFORM_CONFIGURATION_ACTION;

EfiPlatformConfigurationActionNone
The controller specified by ControllerHandle is still in a usable state, its configuration has been updated via parsing the ParameterBlock. If required by the parameter block and the module supports an NVRAM store the configuration information from ParameterBlock was successfully saved to the NVRAM. No actions are required before this controller can be used again with the updated configuration settings.

EfiPlatformConfigurationStopController
The driver has detected that the controller specified by ControllerHandle is not in a usable state, and it needs to be stopped. The calling agent can use the EFI_BOOT_SERVICES.DisconnectController() service to perform this operation, and it should be performed as soon as possible.

EfiPlatformConfigurationRestartController
This controller specified by ControllerHandle needs to be stopped and restarted before it can be used again. The calling agent can use the DisconnectController() and EFI_BOOT_SERVICES.ConnectController() services to perform this operation. The restart operation can be delayed until all of the configuration options have been set.

EfiPlatformConfigurationRestartPlatform
A configuration change has been made that requires the platform to be restarted before the controller specified by ControllerHandle can be used again. The calling agent can use the ResetSystem() services to perform this operation. The restart operation can be delayed until all of the configuration options have been set.

EfiPlatformConfigurationActionNvramFailed
The controller specified by ControllerHandle is still in a usable state; its configuration has been updated via parsing the ParameterBlock. The driver tried to update the driver’s private NVRAM store with information from ParameterBlock and failed. No actions are required before this controller can be used again with the updated configuration settings, but these configuration settings are not guaranteed to persist after ControllerHandle is stopped.
EfiPlatformConfigurationActionUnsupportedGuid

The controller specified by ControllerHandle is still in a usable state; its configuration has not been updated via parsing the ParameterBlock. The driver did not support the ParameterBlock format identified by ParameterTypeGuid. No actions are required before this controller can be used again. On additional Query calls from this ControllerHandle, the platform should stop returning a ParameterBlock qualified by this same ParameterTypeGuid. If no other ParameterTypeGuid is supported by the platform, Query should return EFI_NOT_FOUND.

Status Codes Returned

<table>
<thead>
<tr>
<th>Status Code</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>EFI_SUCCESS</td>
<td>The platform return parameter information for ControllerHandle.</td>
</tr>
<tr>
<td>EFI_NOT_FOUND</td>
<td>Instance was not found.</td>
</tr>
<tr>
<td>EFI_INVALID_PARAMETER</td>
<td>ControllerHandle is NULL.</td>
</tr>
</tbody>
</table>

11.7.1 DMTF SM CLP ParameterTypeGuid

The following parameter protocol ParameterTypeGuid provides the support for parameters communicated through the DMTF SM CLP Specification 1.0 Final Standard to be used to configure the UEFI driver.

In this section the producer of the EFI_PLATFORM_TO_DRIVER_CONFIGURATION_PROTOCOL is platform firmware and the consumer is the UEFI driver.

**Note:** *If future versions of the DMTF SM CLP Specification require changes to the parameter block definition, newer ParameterTypeGuid will be used.*

**GUID**

```c
#define EFI_PLATFORM_TO_DRIVER_CONFIGURATION_CLP_GUID \
{0x34,0ecc,0e,0xcb,06,0x4b,07,05,0x9c,0x47,0x33,0x3e} \
{0xbb, 0x57, 0x1b, 0x12, 0x9c, 0x47, 0x33,0x3e}
```

**Parameter Block**

```c
typedef struct {
  CHAR8  *CLPCommand;
  UINT32 CLPCommandLength;
  CHAR8  *CLPReturnString;
  UINT32 CLPReturnStringLength;
  UINT8  CLPCmdStatus;
  UINT8  CLPErrorValue;
  UINT16 CLPMsgCode;
} EFI_CONFIGURE_CLP_PARAMETER_BLK;
```

**Structure Member Definitions**

- **CLPCommand**: A pointer to the null-terminated UTF-8 string that specifies the DMTF SM CLP command line that the driver is required to parse and process when this function is called. See the DMTF SM CLP
Specification 1.0 Final Standard for details on the format and syntax of the CLP command line string.

CLPCommand buffer is allocated by the producer of the EFI_PLATFORM_TO_DRIVER_CONFIGURATION_PROTOCOL.

CLPCommandLength The length of the CLP Command in bytes.

CLPReturnString A pointer to the null-terminated UTF-8 string that indicates the CLP return status that the driver is required to provide to the calling agent. The calling agent may parse and/or pass this for processing and user feedback. The SM CLP Command Response string buffer is filled in by the UEFI driver in the “keyword=value” format described in the SM CLP Specification (see section 3.table 101, “Output Data”), unless otherwise requested via the SM CLP –output option in the Command Line string buffer. UEFI driver’s support for this default “keyword=value” output format is required if the UEFI driver supports this protocol, while support for other SM CLP output formats is optional. (The UEFI Driver should set CLPCmdStatus=2 (COMMAND PROCESSING FAILED) and CLPErrorValue=249 (OUTPUT FORMAT NOT SUPPORTED) if the SM CLP –output option requested by the caller is not supported by the UEFI Driver.).

CLPReturnString buffer is allocated by the consumer of the EFI_PLATFORM_TO_DRIVER_CONFIGURATION_PROTOCOL and undefined prior to the call to Response().

CLPReturnStringLength The length of the CLP return status string in bytes.

CLPCmdStatus SM CLP Command Status (see DMTF SM CLP Specification 1.0 Final Standard - Table 4)

CLPErrorValue SM CLP Processing Error Value (see DMTF SM CLP Specification 1.0 Final Standard - Table 6).

This field is filled in by the consumer of the EFI_PLATFORM_TO_DRIVER_CONFIGURATION_PROTOCOL and undefined prior to the call to Response().

CLPMsgCode Bit 15: OEM Message Code Flag
0 = Message Code is an SM CLP Probable Cause Value.
(see SM CLP Specification Table 11)
1 = Message Code is OEM Specific

Bits 14-0: Message Code

This field is filled in by the consumer of the EFI_PLATFORM_TO_DRIVER_CONFIGURATION_PROTOCOL and undefined prior to the call to Response().
11.8 EFI Driver Supported EFI Version Protocol

**EFI_DRIVER_SUPPORTED_EFI_VERSION_PROTOCOL**

**Summary**

Provides information about the version of the EFI specification that a driver is following. This protocol is required for EFI drivers that are on PCI and other plug in cards.

**GUID**

```c
#define EFI_DRIVER_SUPPORTED_EFI_VERSION_PROTOCOL_GUID {
  0x5c198761, 0x16a8, 0x4e69,
  { 0x97, 0x2c, 0x89, 0xd6, 0x79, 0x54, 0xf8, 0x1d } }
```

**Protocol Interface Structure**

```c
typedef struct EFI_DRIVER_SUPPORTED_EFI_VERSION_PROTOCOL {
  UINT32 Length;
  UINT32 FirmwareVersion;
} EFI_DRIVER_SUPPORTED_EFI_VERSION_PROTOCOL;
```

**Parameters**

- **Length**
  
  The size, in bytes, of the entire structure. Future versions of this specification may grow the size of the structure.

- **FirmwareVersion**
  
  The latest version of the UEFI Specification that this driver conforms to. Refer to the `EFI_SPECIFICATION_VERSION` definition in Section 4.3.

**Description**

The `EFI_DRIVER_SUPPORTED_EFI_VERSION_PROTOCOL` provides a mechanism for an EFI driver to publish the version of the EFI specification it conforms to. This protocol must be placed on the drivers image handle when the driver's entry point is called.

11.9 EFI Driver Family Override Protocol

11.9.1 Overview

This section defines the Driver Family Override Protocol, and contains the following:

- Description and code definitions of the Driver Family Override Protocol.
- Required updates to the EFI Boot Services ConnectController().
- Typical production of the Driver Family Override Protocol by an EFI Driver that follows the EFI Driver Model.

The Driver Family Override Protocol provides a method for an EFI Driver to opt-in to a higher priority rule for connecting drivers to controllers in the EFI Boot Service `ConnectController()`. This new rule is higher priority than the Bus Specific Driver Override Protocol rule and lower priority than the Platform Driver Override Rule.
The Driver Family Override Protocol is a backwards compatible extension to the EFI Driver Model and is only available during boot time. The Driver Family Override Protocol may be optionally produced by a driver that follows the EFI Driver Model. If this protocol is produced, it must be installed onto the Driver Image Handle. Drivers that follow the EFI Driver Model typically install the EFI Driver Binding Protocol onto the driver's image handle. In this case, the Driver Family Override Protocol must also be installed onto the driver's image handle. If a single EFI Driver produces more than one instance of the EFI Driver Binding Protocol, then the Driver Family Override Protocol must be installed onto the same handle as the EFI Driver Binding Protocol that is associated with the Driver Family Override Protocol. Since it is legal for a single EFI Driver to produce multiple EFI Driver Binding Protocol instances, it is also legal for a single EFI Driver to produce multiple Driver Family Override Protocol instances.

**EFI_DRIVER_FAMILY_OVERRIDE_PROTOCOL**

**Summary**

When installed, the Driver Family Override Protocol informs the UEFI Boot Service `ConnectController()` that this driver is higher priority than the list of drivers returned by the Bus Specific Driver Override Protocol.

**GUID**

```c
#define EFI_DRIVER_FAMILY_OVERRIDE_PROTOCOL_GUID \
  {0xb1ee129e,0xda36,0x4181,\ 
   {0x91,0xf8,0x04,0xa4,0x92,0x37,0x66,0xa7}}
```

**Protocol Interface Structure**

```c
typedef struct _EFI_DRIVER_FAMILY_OVERRIDE_PROTOCOL {
  EFI_DRIVER_FAMILY_OVERRIDE_GET_VERSION GetVersion;
} EFI_DRIVER_FAMILY_OVERRIDE_PROTOCOL;
```

**Parameters**

- **GetVersion**
  
  Retrieves the version of the driver that is used by the EFI Boot Service `ConnectController()` to sort the set of Driver Binding Protocols in order from highest priority to lowest priority. For drivers that support the Driver Family Override Protocol, those drivers are sorted so that the drivers with higher values returned by `GetVersion()` are high priority that drivers that return lower values from `GetVersion()`.

**Description**

This protocol contains a single service that returns a version value for the driver that produces this protocol. High values are higher priority than lower values when evaluated by the EFI Boot Service `ConnectController()`. This is an optional protocol that may be produced by an EFI Driver that follows the EFI Driver Model. If this protocol is produced, it must be installed onto a handle that also contains the EFI Driver Binding Protocol.

If this protocol is not produced by an EFI Driver, then the rules used to connect a driver to a controller from highest priority to lowest priority are as follows:

- **Context Override**
• Platform Driver Override
• Bus Specific Driver Override Protocol
• Driver Binding Search

If this protocol is produced by an EFI Driver, then the rules used to connect a driver to a controller from highest priority to lowest priority are as follows:

• Context Override
• Platform Driver Override
• Driver Family Override
• Bus Specific Driver Override
• Driver Binding Search

**EFI_DRIVER_FAMILY_OVERRIDE_PROTOCOL.GetVersion ()**

**Summary**
Retrieves the version of the driver that is used by the EFI Boot Service `ConnectController()` to sort the set of Driver Binding Protocols in order from highest priority to lowest priority. For drivers that support the Driver Family Override Protocol, those drivers are sorted so that the drivers with higher values returned by `GetVersion()` are high priority that drivers that return lower values from `GetVersion()`.

**Prototype**

```c
typedef UINT32 (EFIAPI *EFI_DRIVER_FAMILY_OVERRIDE_GET_VERSION) (IN EFI_DRIVER_FAMILY_OVERRIDE_PROTOCOL *This);
```

**Parameters**

*This*
A pointer to the `EFI_DRIVER_FAMILY_OVERRIDE_PROTOCOL` instance.

**Description**
This function returns the version value associated with the driver specified by *This*.

### 11.10 EFI Driver Health Protocol

This section contains the basic definitions of the Driver Health Protocol.
**EFI_DRIVER_HEALTH_PROTOCOL**

**Summary**
When installed, the Driver Health Protocol produces a collection of services that allow the health status for a controller to be retrieved. If a controller is not in a usable state, status messages may be reported to the user, repair operations can be invoked, and the user may be asked to make software and/or hardware configuration changes. All display, as well as interaction, with the user must be handled by the consumer of the Driver Health Protocol.

The Driver Health Protocol must be installed onto the same handle as the associated Driver Binding handle.

**GUID**

```c
#define EFI_DRIVER_HEALTH_PROTOCOL_GUID \ 
  {0x2a534210,0x9280,0x41d8,\ 
    {0xae,0x79,0xca,0xda,0x01,0xa2,0xb1,0x27 }}
```

**Protocol Interface Structure**

```c
typedef struct _EFI_DRIVER_HEALTH_PROTOCOL {
  EFI_DRIVER_HEALTH_GET_HEALTH_STATUS GetHealthStatus;
  EFI_DRIVER_HEALTH_REPAIR Repair;
} EFI_DRIVER_HEALTH_PROTOCOL;
```

**Parameters**
- **GetHealthStatus**
  Retrieves the health status of a controller in the platform. This function can also optionally return warning messages, error messages, and an HII Form that may be used to repair a controller that is not properly configured.

- **Repair**
  Performs a repair operation on a controller in the platform. This function can optionally report repair progress information back to the platform.

**Description**
The Driver Health Protocol is optionally produced by a driver that follows the EFI Driver Model. If an EFI Driver needs to report health status to the platform, provide warning or error messages to the user, perform length repair operations, or request the user to make hardware or software configuration changes, then the Driver Health Protocol must be produced.

A controller that is managed by a driver that follows the EFI Driver Model and produces the Driver Health Protocol must report the current health of the controllers that the driver is currently managing. The controller can initially be healthy, failed, require repair, or require configuration. If a controller requires configuration, and the user make configuration changes, the controller may then need to be reconnected or the system may need to be rebooted for the configuration changes to take effect. Figure 2-1 below shows all the possible health states of a controller, the set of initial states, the set of terminal states, and the legal transitions between the health states.
Figure 11-1 Driver Health Status States

**EFI_DRIVER_HEALTH_PROTOCOL.GetHealthStatus()**

**Summary**

Retrieves the health status of a controller in the platform. This function can also optionally return warning messages, error messages, and an HII Form that may be used to repair a controller that is not properly configured.

**Prototype**

```c
typedef
    EFI_STATUS
    (EFIAPI *EFI_DRIVER_HEALTH_GET_HEALTH_STATUS) (
        IN EFI_DRIVER_HEALTH_PROTOCOL *This,
        IN EFI_HANDLE ControllerHandle, OPTIONAL
        IN EFI_HANDLE ChildHandle, OPTIONAL
        OUT EFI_DRIVER_HEALTH_STATUS *HealthStatus,
        OUT EFI_DRIVER_HEALTH_HII_MESSAGE **MessageList,  OPTIONAL
        OUT EFI_HII_HANDLE *FormHiiHandle OPTIONAL
    );
```

**Parameters**

- **This**
  A pointer to the **EFI_DRIVER_HEALTH_PROTOCOL** instance.
ControllerHandle  The handle of the controller to retrieve the health status on. This is an optional parameter that may be NULL. If this parameter is NULL, then the value of ChildHandle is ignored, and the combined health status of all the devices that the driver is managing is returned.

ChildHandle  The handle of the child controller to retrieve the health status on. This is an optional parameter that may be NULL. It will be NULL for device drivers. It will also be NULL for bus drivers when an attempt is made to collect the health status of the bus controller. If will not be NULL when an attempt is made to collect the health status for a child controller produced by the driver. If ControllerHandle is NULL, then this parameter is ignored.

HealthStatus  A pointer to the health status that is returned by this function. The health status for the controller specified by ControllerHandle and ChildHandle is returned.

MessageList  A pointer to an array of warning or error messages associated with the controller specified by ControllerHandle and ChildHandle. This is an optional parameter that may be NULL. MessageList is allocated by this function with the EFI Boot Service AllocatePool(), and it is the caller’s responsibility to free MessageList with the EFI Boot Service FreePool(). Each message is specified by tuple of an EFI_HII_HANDLE and an EFI_STRING_ID. The array of messages is terminated by tuple containing a EFI_HII_HANDLE with a value of NULL. The EFI_HII_STRING_PROTOCOL.GetString() function can be used to retrieve the warning or error message as a Null-terminated string in a specific language. Messages may be returned for any of the HealthStatus values except EfiDriverHealthStatusReconnectRequired and EfiDriverHealthStatusRebootRequired.

FormHiiHandle  A pointer to the HII handle containing the HII form used when configuration is required. The HII handle is associated with the controller specified by ControllerHandle and ChildHandle. If this is NULL, then no HII form is available. An HII handle will only be returned with a HealthStatus value of EfiDriverHealthStatusConfigurationRequired.

Description
This function returns the health status associated with the controller specified by ControllerHandle and ChildHandle. If ControllerHandle is NULL, and if the EFI driver is not managing any controller then EfiDriverHealthStatusFailed is returned in HealthStatus and EFI_DEVICE_ERROR is returned. If ControllerHandle is not NULL and the driver specified by This is not currently managing the controller specified by ControllerHandle and ChildHandle, then EFI_UNSUPPORTED is returned. If HealthStatus is NULL, then EFI_INVALID_PARAMETER is returned.

If ControllerHandle is NULL, then the cumulative health status of all the controllers managed by the EFI driver is returned. If all the controllers managed by the driver are healthy, then EfiDriverHealthStatusHealthy must be returned in HealthStatus. If one or more of the
controllers managed by the EFI Driver is not healthy, then `EfiDriverHealthStatusFailed` must be returned.

If `ControllerHandle` is not `NULL` and `ChildHandle` is `NULL`, then the health status of the controller specified by `ControllerHandle` is returned in `HealthStatus` and `EFI_SUCCESS` is returned.

If `ControllerHandle` is not `NULL` and `ChildHandle` is not `NULL`, then the health status of the child controller specified by `ControllerHandle` and `ChildHandle` is returned in `HealthStatus` and `EFI_SUCCESS` is returned.

If `MessageList` is `NULL`, then no messages are returned from this function.

If `MessageList` is not `NULL`, and `HealthStatus` is `EfiDriverHealthStatusReconnectRequired` or `EfiDriverHealthStatusRebootRequired` then no messages are returned and `MessageList` must be set to `NULL`.

If `MessageList` is not `NULL`, and there are no warning or error messages associated with the controller specified by `ControllerHandle` and `ChildHandle`, then `MessageList` must be set to `NULL`.

If `MessageList` is not `NULL`, and there are one or more warning or error messages associated with the controller specified by `ControllerHandle` and `ChildHandle`, then `MessageList` must point to a buffer allocated with the EFI Boot Service `AllocatePool()`. The number of `EFI_DRIVER_HEALTH_HII_MESSAGE` structures allocated in the buffer must be one more than the total number of warning or error messages, and the `HiiHandle` field of the last `EFI_DRIVER_HEALTH_HII_MESSAGE` structure must be set to `NULL` to terminate the list of messages.

It is the caller’s responsibility to free the buffer returned in `MessageList` using the EFI Boot Service `FreePool()`. Each message is specified by an `EFI_HII_HANDLE` and an `EFI_STRING_ID`. The caller may use the `EFI_HII_STRING_PROTOCOL.GetString()` function to convert each message into a null-terminated string that can be displayed on a console device.

If `FormHiiHandle` is `NULL`, then no forms are returned from this function.

If `FormHiiHandle` is not `NULL`, and `HealthStatus` is not `EfiDriverHealthStatusConfigurationRequired`, then no forms are returned and `FormHiiHandle` must be set to `NULL`.

If `FormHiiHandle` is not `NULL`, and `FormSetGuid` is not `NULL`, and `HealthStatus` is `EfiDriverHealthStatusConfigurationRequired`, then `FormHiiHandle` is assigned to the HII handle which contains the HII form required to perform the configuration operation.

If `ControllerHandle` is `NULL`, and there are no devices being managed by the driver then `EFI_UNSUPPORTED` is returned.
Related Definitions

```
//*******************************************************************************
// EFI_DRIVER_HEALTH_STATUS
//*******************************************************************************
typedef enum {
    EfiDriverHealthStatusHealthy,
    EfiDriverHealthStatusRepairRequired,
    EfiDriverHealthStatusConfigurationRequired,
    EfiDriverHealthStatusFailed,
    EfiDriverHealthStatusReconnectRequired,
    EfiDriverHealthStatusRebootRequired
} EFI_DRIVER_HEALTH_STATUS;
```

**EfiDriverHealthStatusHealthy**

The controller is in a healthy state.

**EfiDriverHealthStatusRepairRequired**

The controller requires a repair operation that will take an extended period of time to perform. The EFI Boot Manager is required to call the `Repair()` function when this state is detected. After the `Repair()` function completed, the health status may be `EfiDriverHealthStatusHealthy`, `EfiDriverHealthStatusConfigurationRequired`, or `EfiDriverHealthStatusFailed`.

**EfiDriverHealthStatusConfigurationRequired**

The controller requires the user to make software or hardware configuration changes in order to put the controller into a healthy state. The set of software configuration changes are specified by the `FormHiiHandle` and `FormSetGuid` parameters. The EFI Boot Manager may call the `EFI_FORM_BROWSER2_PROTOCOL.SendForm()` function to display configuration information and allow the user to make the required configuration changes. The HII form is the first enabled form in the form set class `EFI_HII_DRIVER_HEALTH_FORMSET_GUID`, which is installed on the returned HII handle `FormHiiHandle`. The `MessageList` parameter may be used to identify additional user configuration operations required to place the controller in a healthy state. After the `FormHiiHandle` and `MessageList` have been processed by the EFI Boot Manager, the health status may be `EfiDriverHealthStatusHealthy`, `EfiDriverHealthStatusConfigurationRequired`, `EfiDriverHealthStatusRepairRequired`, `EfiDriverHealthStatusFailed`, `EfiDriverHealthStatusReconnectRequired`, or `EfiDriverHealthStatusRebootRequired`.

**EfiDriverHealthStatusFailed**

The controller is in a failed state, and there no actions that can place the controller into a healthy state. This controller can not be used as a boot device and no boot devices behind this controller can be used as a boot device.
**EfiDriverHealthStatusReconnectRequired**

A hardware and/or software configuration change was performed by the user, and the controller needs to be reconnected before the controller can be placed in a healthy state. The EFI Boot Manager is required to call the EFI Boot Service `DisconnectController()` followed by the EFI Boot Service `ConnectController()` to reconnect the controller.

**EfiDriverHealthStatusRebootRequired**

A hardware and/or software configuration change was performed by the user, and the controller requires the entire platform to be rebooted before the controller can be placed in a healthy state. The EFI Boot Manager should complete the configuration and repair operations on all the controllers that are not in a healthy state before rebooting the system.

```c
typedef struct {
    EFI_HII_HANDLE HiiHandle;
    EFI_STRING_ID StringId;
    UINT64 MessageCode;
} EFI_DRIVER_HEALTH_HII_MESSAGE;
```

- **HiiHandle**: The `EFI_HII_HANDLE` that was returned by `EFI_HII_DATABASE_PROTOCOL.NewPackageList()` when the string pack containing `StringId` was registered with the HII Database.

- **StringId**: The identifier for a single string token in the string pack associated with `HiiHandle`.

- **MessageCode**: 64-bit numeric value of the warning/error specified by this message. A value of `0x0000000000000000` is used to indicate that `MessageCode` is not specified.

  The values `0x0000000000000001` to `0xfffffffffffffff` are reserved for allocation by the UEFI Specification.

  The values `0x1000000000000000` to `0x1fffffffffffffff` are reserved for IHV-developed drivers.

  The values `0x8000000000000000` to `0x8fffffffffffffff` is reserved for platform/OEM drivers.

  All other values are reserved and should not be used.
Status Codes Returned

<table>
<thead>
<tr>
<th>Code</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>EFI_SUCCESS</td>
<td>The health status of the controller specified by ControllerHandle and ChildHandle was returned in HealthStatus. A list of warning and error messages may be optionally returned in MessageList, and an HII Form may be optionally specified by FormHiiHandle.</td>
</tr>
<tr>
<td>EFI_UNSUPPORTED</td>
<td>ControllerHandle is not NULL, and the controller specified by ControllerHandle and ChildHandle is not currently being managed by the driver specified by This.</td>
</tr>
<tr>
<td>EFI_UNSUPPORTED</td>
<td>ControllerHandle is NULL and there are no devices being managed by the driver.</td>
</tr>
<tr>
<td>EFI_INVALID_PARAMETER</td>
<td>HealthStatus is NULL.</td>
</tr>
<tr>
<td>EFI_OUT_OF_RESOURCES</td>
<td>MessageList is not NULL, and there are not enough resource available to allocate memory for MessageList.</td>
</tr>
</tbody>
</table>

EFI_DRIVER_HEALTH_PROTOCOL.Repair()

Summary
Performs a repair operation on a controller in the platform. This function can optionally report repair progress information back to the platform.

Prototype
```
typedef EFI_STATUS (EFIAPIC *EFI_DRIVER_HEALTH_REPAIR) (  
    IN EFI_DRIVER_HEALTH_PROTOCOL *This,  
    IN EFI_HANDLE ControllerHandle,  
    IN EFI_HANDLE ChildHandle OPTIONAL,  
    IN EFI_DRIVER_HEALTH_REPAIR_NOTIFY RepairNotify OPTIONAL  
);
```

Parameters
- **This**: A pointer to the EFI_DRIVER_HEALTH_PROTOCOL instance.
- **ControllerHandle**: The handle of the controller to repair.
- **ChildHandle**: The handle of the child controller to repair. This is an optional parameter that may be NULL. It will be NULL for device drivers. It will also be NULL for bus drivers when an attempt is made to repair a bus controller. If will not be NULL when an attempt is made to repair a child controller produced by the driver.
- **RepairNotify**: A notification function that may be used by a driver to report the progress of the repair operation. This is an optional parameter that may be NULL.

Description
This function repairs the controller specified by ControllerHandle and ChildHandle. If the driver specified by This is not currently managing the controller specified by ControllerHandle and
ChildHandle, then **EFI_UNSUPPORTED** is returned. If there are not enough resource available to complete the repair operation, then **EFI_OUT_OF_RESOURCES** is returned. Otherwise, **EFI_SUCCESS** is returned. A return value of **EFI_SUCCESS** does not guarantee that the controller is in a healthy state. The EFI Boot Manager must call the `GetHealthStatus()` function to determine the result of the repair operation.

If `RepairNotify` is not **NULL**, and the repair operation requires an extended period of time to execute, then the driver performing the repair operation may intermittently call the `RepairNotify` function to inform the EFI Boot Manager of the progress of the repair operation. The `RepairNotify` function takes two parameters to specify the current progress value and the limit value. These two values may be used by the EFI Boot Manager to present status information for the current repair operation.

### Related Definitions

```
//******************************************************************************
// EFI_DRIVER_HEALTH_REPAIR_NOTIFY
//******************************************************************************
typedef
EFI_STATUS
(EFIAPI *EFI_DRIVER_HEALTH_REPAIR_NOTIFY)(
    IN UINTN Value,
    IN UINTN Limit
);
```

- **Value**: A value between 0 and Limit that identifies the current progress of the repair operation.
- **Limit**: The maximum value of Value for the current repair operation. If Limit is 0, then the completion progress is indeterminate. For example, a driver that wants to specify progress in percent would use a Limit value of 100.

### Status Codes Returned

<table>
<thead>
<tr>
<th>Code</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>EFI_SUCCESS</strong></td>
<td>An attempt to repair the controller specified by <code>ControllerHandle</code> and <code>ChildHandle</code> was performed. The result of the repair operation can be determined by calling <code>GetHealthStatus()</code>.</td>
</tr>
<tr>
<td><strong>EFI_UNSUPPORTED</strong></td>
<td>The driver specified by This is not currently managing the controller specified by <code>ControllerHandle</code> and <code>ChildHandle</code>.</td>
</tr>
<tr>
<td><strong>EFI_OUT_OF_RESOURCES</strong></td>
<td>There are not enough resources to perform the repair operation.</td>
</tr>
</tbody>
</table>

### 11.10.1 UEFI Boot Manager Algorithms

This section contains example algorithms that a UEFI Boot Manager or UEFI Application could use to interact with one or more instances of the Efi Driver Health Protocol present in the platform.
11.10.1.1 All Controllers Healthy

This section contains example algorithms that a UEFI Boot Manager or UEFI Application could use to interact with one or more instances of the EFI Driver Health Protocol present in the platform.

The following algorithm collects all the EFI Driver Health Protocols currently present in the EFI Handle Database, and queries each EFI Driver Health Protocol to determine if one or more of the controllers managed by each EFI Driver Health Protocol instance are not healthy. The variable `AllHealthy` is TRUE if all the controllers in the platform are healthy. `AllHealthy` is FALSE if one or more of the controllers in the platform are not healthy.

```c
EFI_STATUS                  Status;
UINTN                      NoHandles;
EFI_HANDLE                 *Handles;
UINTN                      Index;
EFI_DRIVER_HEALTH_PROTOCOL *DriverHealth;
BOOLEAN                    AllHealthy;

Status = gBS->LocateHandleBuffer (
    ByProtocol,  
    &gEfiDriverHealthProtocolGuid,  
    NULL,  
    &NoHandles,  
    &Handles
);
if (EFI_ERROR (Status)) {
    return;
}

AllHealthy = TRUE;
for (Index = 0; Index < NoHandles; Index++) {
    Status = gBS->HandleProtocol (
        Handles[Index],  
        &gEfiDriverHealthProtocolGuid,  
        (VOID **)&DriverHealth
    );
    if (!EFI_ERROR (Status)) {
        Status = DriverHealth->GetHealthStatus (
            DriverHealth,  
            NULL,  
            NULL,  
            NULL,  
            NULL,  
            NULL,  
            NULL
        );
    }
}
```
if (EFI_ERROR (Status)) {
    AllHealthy = FALSE;
}

11.10.1.2 Process a Controller Until Terminal State Reached

The following algorithm processes a single controller using the EFI Driver Health Protocol associated with that controller. This algorithm continues to query the `GetHealthStatus()` service until one of the legal terminal states of the EFI Driver Health Protocol is reached. This may require the processing of HII Messages, HII Form, and invocation of repair operations.

```c
EFI_STATUS Status;
EFI_DRIVER_HEALTH_PROTOCOL *DriverHealth;
EFI_HANDLE ControllerHandle;
EFI_HANDLE ChildHandle;
EFI_DRIVER_HEALTH_HEALTH_STATUS HealthStatus;
EFI_DRIVER_HEALTH_HII_MESSAGE *MessageList;
EFI_HII_HANDLE FormHiiHandle;

do {
    HealthStatus = EfiDriverHealthStatusHealthy;
    Status = DriverHealth->GetHealthStatus (
        DriverHealth,
        ControllerHandle,
        ChildHandle,
        &HealthStatus,
        &MessageList,
        &FormHiiHandle
    );
    ProcessMessages (MessageList);
    if (HealthStatus == EfiDriverHealthStatusRepairRequired) {
        Status = DriverHealth->Repair (
            DriverHealth,
            ControllerHandle,
            ChildHandle,
            RepairNotify
        );
    }
    if (HealthStatus == EfiDriverHealthStatusConfigurationRequired) {
        ProcessForm (FormHiiHandle);
    }
} while (HealthStatus == EfiDriverHealthStatusConfigurationRequired ||
        HealthStatus == EfiDriverHealthStatusRepairRequired);

//
// Check for RebootRequired or ReconnectRequired
//
11.10.1.3 Repair Notification Function

The following is an example repair notification function.

```c
VOID
RepairNotify ( 
    UINTN Value, 
    UINTN Limit 
) 
{
    UINTN Percent;

    if (Limit == 0) {
        Print (L"Repair Progress Undefined\n\r");
    } else {
        Percent = Value * 100 / Limit;
        Print (L"Repair Progress = %3d%%", Percent);
    }
}
```

11.10.1.4 Process Message List

The following algorithm processes a set of messages returned by the `GetHealthStatus()` service of the EFI Driver Health Protocol.

```c
EFI_STATUS Status;
EFI DRIVER_HEALTH_HII_MESSAGE *MessageList;
UINTN MessageIndex;
EFI_HII_STRING_PROTOCOL *HiiString;
EFI_STRING MessageString[200];

for (MessageIndex = 0; 
    MessageList[MessageIndex].HiiHandle != 0; 
    MessageIndex++) {
    MessageLength = sizeof (MessageString);
    Status = HiiString->GetString ( 
        HiiString, 
        NULL, 
        MessageList[MessageIndex].HiiHandle, 
        MessageList[MessageIndex].StringId, 
        MessageString 
        &MessageLength, 
        NULL 
    );
    if (!EFI_ERROR (Status)) {
        // Log or Print or Display MessageString
    }
}
11.10.1.5 Process HII Form

The following algorithm processes an HII Form returned by the `GetHealthStatus()` service of the EFI Driver Health Protocol.

```c
EFI_STATUS Status;
EFI_FORM_BROWSER2_PROTOCOL *FormBrowser;
EFI_HII_HANDLE FormHiiHandle;

Status = FormBrowser->SendForm (FormBrowser,
                                &FormHiiHandle,
                                1,
                                &gEfiHiiDriverHealthFormsetGuid,
                                0,
                                NULL,
                                NULL);
```

11.10.2 UEFI Driver Algorithms

A UEFI Driver that supports the EFI Driver Health Protocol will typically make the following changes:

11.10.2.1 Driver Entry Point Updates

Install Driver Health Protocol on the driver image handle.

Register HII String/IFR packs with the HII Database

- HII String/IFR packs can also be carried in a PE/COFF image extension eliminating the need for the driver to perform the registration
- The HII String and HII Forms may be produced dynamically when the `GetHealthStatus()` service is called.

11.10.2.2 Add global variable

Add global variable to track combined health status of all controllers managed by the driver. The variable is TRUE if all the controllers managed by the driver are healthy. The variable is FALSE if one or more controllers managed by the driver are not healthy.

11.10.2.3 Update private context structure

Update private context structure to track health status of each controller managed by the driver. This may also include the current set of HII Strings and HII Forms associated with the controllers that are not healthy.
11.10.2.4 Implement GetHealthStatus() service
Implement GetHealthStatus() service of the EFI Driver Health Protocol

- Make sure only legal state transitions are implemented
- Evaluate configuration data and repair status
- Return HII Strings for message(s) associated with the current state
- If configuration required, return HII Form to be processed

11.10.2.5 Implement Repair() service
Implement Repair() service of the EFI Driver Health Protocol

- Calling Repair Notification callback is optional, but recommended.
- Update health status in private context structure before returning
- Make sure only legal state transitions are implemented

11.11 EFI Adapter Information Protocol
This section provides a detailed description of the EFI_ADAPTER_INFORMATION_PROTOCOL. The EFI Adapter Information Protocol is used to dynamically and quickly discover or set device information for an adapter. The discovery of information and state of an adapter should be quick and only return dynamic information. The information should never be cached or stale. The setting information for the adapter should also be fast and simple. The only information that should be set is operating state information, like setting a speed. This protocol is meant to be light weight and non-blocking.

**EFI_ADAPTER_INFORMATION_PROTOCOL**

**SUMMARY**

Since this protocol will return and set information for the adapter, the adapter device driver must publish the EFI_ADAPTER_INFORMATION_PROTOCOL.

There are many kinds of adapters. The set and get adapter information functions should be used to determine the current state of the adapter, or to set a state for an adapter, like device speed.

**GUID**

```
#define EFI_ADAPTER_INFORMATION_PROTOCOL_GUID
{ 0xE5DD1403, 0xD622, 0xC24E, 
  { 0x84, 0x88, 0xC7, 0x1B, 0x17, 0xF5, 0xE8, 0x02 } }
```

**Protocol Interface Structure**

```
typedef struct _EFI_ADAPTER_INFORMATION_PROTOCOL {
    EFI_ADAPTER_INFO_GET_INFO    GetInformation;
    EFI_ADAPTER_INFO_SET_INFO    SetInformation;
    EFI_ADAPTER_INFO_GET_SUPPORTED_TYPES GetSupportedTypes;
} EFI_ADAPTER_INFORMATION_PROTOCOL;
```

**Parameters**
GetInformation          Gets device state information from adapter. See GetInformation() for more function description.
SetInformation         Sets device information for adapter. See SetInformation() for more function description.
GetSupportedTypes      Gets a list of supported information types for this instance of the protocol.

Description
The EFI_ADAPTER_INFORMATION_PROTOCOL is used to get or set the state for an adapter.

EFI_ADAPTER_INFORMATION_PROTOCOL. EFI_ADAPTER_GET_INFO()

Summary
Returns the current state information for the adapter.

Prototype

typedef
  EFI_STATUS
  (EFIAPI *EFI_ADAPTER_INFO_GET_INFO) (  
    IN   EFI_ADAPTER_INFORMATION_PROTOCOL *This,
    IN   EFI_GUID                         *InformationType,
    OUT  VOID                             **InformationBlock,
    OUT  UINTN                            *InformationBlockSize
  );

Parameters
This
A pointer to the EFI_ADAPTER_INFORMATION_PROTOCOL instance.

InformationType
A pointer to an EFI_GUID that defines the contents of InformationBlock. The caller must use the InformationType to specify the information it needs to retrieve from this service and to determine how to parse the InformationBlock. The driver should not attempt to free InformationType.

InformationBlock
This service returns a pointer to the buffer with the InformationBlock structure which contains details about the data specific to InformationType. This structure is defined based on the type of data returned, and will be different for different data types. This service and caller decode this structure and its contents based on InformationType. This buffer is allocated by this service, and it is the responsibility of the caller to free it after using it. Ignored if InformationBlockSize is 0.

InformationBlockSize
The driver returns the size of the InformationBlock in bytes.
Description

The `GetInformation()` function returns information of type `InformationType` from the adapter. If an adapter does not support the requested informational type, then `EFI_UNSUPPORTED` is returned. If an adapter does not contain information for the requested `InformationType`, it fills `InformationBlockSize` with 0 and returns `EFI_NOT_FOUND`.

Status Codes Returned

<table>
<thead>
<tr>
<th>Status Code</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>EFI_SUCCESS</td>
<td>The <code>InformationType</code> information was retrieved.</td>
</tr>
<tr>
<td>EFI_UNSUPPORTED</td>
<td>The <code>InformationType</code> is not known.</td>
</tr>
<tr>
<td>EFI_NOT_FOUND</td>
<td>Information is not available for the requested information type.</td>
</tr>
<tr>
<td>EFI_DEVICE_ERROR</td>
<td>The device reported an error.</td>
</tr>
<tr>
<td>EFI_OUT_OF_RESOURCES</td>
<td>The request could not be completed due to a lack of resources</td>
</tr>
<tr>
<td>EFI_INVALID_PARAMETER</td>
<td>This is NULL</td>
</tr>
<tr>
<td>EFI_INVALID_PARAMETER</td>
<td>InformationBlock is NULL</td>
</tr>
<tr>
<td>EFI_INVALID_PARAMETER</td>
<td>InformationBlockSize is NULL</td>
</tr>
</tbody>
</table>

`EFI_ADAPTER_INFORMATION_PROTOCOL. EFI_ADAPTER_INFO_SET_INFO()`

Summary

Sets state information for an adapter.

Prototype

```c
typedef
EFI_STATUS
(EFI_API *EFI_ADAPTER_INFO_SET_INFO) (  
  IN EFIAPI *EFI_ADAPTER_INFORMATION_PROTOCOL *This,  
  IN EFI_GUID *InformationType,  
  IN VOID *InformationBlock,  
  IN UINTN InformationBlockSize  
);
```

Parameters

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>This</td>
<td>A pointer to the <code>EFI_ADAPTER_INFORMATION_PROTOCOL</code> instance.</td>
</tr>
<tr>
<td>InformationType</td>
<td>A pointer to an <code>EFI_GUID</code> that defines the contents of <code>InformationBlock</code>.</td>
</tr>
<tr>
<td>InformationBlockSize</td>
<td>The caller must use the <code>InformationType</code> to specify the information it wants the service.</td>
</tr>
<tr>
<td>InformationBlock</td>
<td>A pointer to the <code>InformationBlock</code> structure which contains details about the data specific to <code>InformationType</code>. This structure is defined based on the type of data sent, and will be different for different data types. The driver and caller decode this structure and...</td>
</tr>
</tbody>
</table>
its contents based on **InformationType**. This buffer is allocated by
the caller. It is the responsibility of the caller to free it after the caller
has set the requested parameters.

**InformationBlockSize** The size of the **InformationBlock** in bytes.

**Description**

The **SetInformation()** function sends information of type **InformationType** for an adapter.
If an adapter does not support the requested informational type, then **EFI_UNSUPPORTED** is returned.

**Related Definitions**

**Status Codes Returned**

<table>
<thead>
<tr>
<th>Status Code</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>EFI_SUCCESS</strong></td>
<td>The information was received and interpreted successfully.</td>
</tr>
<tr>
<td><strong>EFI_UNSUPPORTED</strong></td>
<td>The <strong>InformationType</strong> is not known.</td>
</tr>
<tr>
<td><strong>EFI_DEVICE_ERROR</strong></td>
<td>The device reported an error.</td>
</tr>
<tr>
<td><strong>EFI_INVALID_PARAMETER</strong></td>
<td><strong>This</strong> is <strong>NULL</strong></td>
</tr>
<tr>
<td><strong>EFI_INVALID_PARAMETER</strong></td>
<td><strong>InformationBlock</strong> is <strong>NULL</strong></td>
</tr>
<tr>
<td><strong>EFI_WRITE_PROTECTED</strong></td>
<td>The <strong>InformationType</strong> cannot be modified using <strong>EFI_ADAPTER_INFO_SET_INFO()</strong></td>
</tr>
</tbody>
</table>

**EFI_ADAPTER_INFORMATION_PROTOCOL.**

**EFI_ADAPTER_INFO_GET_SUPPORTED_TYPES()**

**Summary**

Get a list of supported information types for this instance of the protocol.

**Prototype**

```c
typedef
EFI_STATUS
(EIFIAPI *EFI_ADAPTER_INFO_GET_SUPPORTED_TYPES) (  
   IN  EFI_ADAPTER_INFORMATION_PROTOCOL *This,
   OUT EFI_GUID                        **InfoTypesBuffer,
   OUT UINTN                           *InfoTypesBufferCount
);  
```

**Parameters**

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>This</strong></td>
<td>A pointer to the <strong>EFI_ADAPTER_INFORMATION_PROTOCOL</strong> instance.</td>
</tr>
<tr>
<td><strong>InfoTypesBuffer</strong></td>
<td>A pointer to the array of <strong>InformationType</strong> GUIDs that are supported by <strong>This</strong>. This buffer is allocated by this service, and it is the responsibility of the caller to free it after using it</td>
</tr>
<tr>
<td><strong>InfoTypesBufferCount</strong></td>
<td>A pointer to the number of GUIDs present in <strong>InfoTypesBuffer</strong>.</td>
</tr>
</tbody>
</table>
Description

The `GetSupportedTypes()` function returns a list of `InformationType` GUIDs that are supported on an adapter with this instance of `EFI_ADAPTER_INFORMATION_PROTOCOL`. The list is returned in `InfoTypesBuffer`, and the number of GUID pointers in `InfoTypesBuffer` is returned in `InfoTypesBufferCount`.

Status Codes Returned

<table>
<thead>
<tr>
<th>Code</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>EFI_SUCCESS</td>
<td>The list of information type GUIDs that are supported on this adapter was returned in <code>InfoTypesBuffer</code>. The number of information type GUIDs was returned in <code>InfoTypesBufferCount</code>.</td>
</tr>
<tr>
<td>EFI_INVALID_PARAMETER</td>
<td>This is <strong>NULL</strong></td>
</tr>
<tr>
<td>EFI_INVALID_PARAMETER</td>
<td><code>InfoTypesBuffer</code> is <strong>NULL</strong></td>
</tr>
<tr>
<td>EFI_INVALID_PARAMETER</td>
<td><code>InfoTypesBufferCount</code> is <strong>NULL</strong></td>
</tr>
<tr>
<td>EFI_OUT_OF_RESOURCES</td>
<td>There is not enough pool memory to store the results</td>
</tr>
</tbody>
</table>

11.12 EFI Adapter Information Protocol Information Types

**Note:** *In addition to the information block types defined in this section, driver writers may define additional information type blocks for their own use provided all such blocks are each identified by a unique GUID created by the definer.*

Clients of the protocol should ignore any unrecognized block types returned by `GetSupportedTypes()`.

11.12.1 Network Media State

For network adapters, the `EFI_ADAPTER_INFORMATION_PROTOCOL` must be installed on the same handle as the UNDI protocol. If SNP or MNP protocol, instead of the UNDI protocol, is installed on adapter handle, then the `EFI_ADAPTER_INFORMATION_PROTOCOL` must be installed on the same handle as the SNP or MNP protocol.

**InformationType**

```c
#define EFI_ADAPTER_INFO_MEDIA_STATE_GUID \
{0xD7C74207, 0xA831, 0x4A26 \ 
{0xB1,0xF5,0xD1,0x93,0x06,0x5C,0xE8,0xB6}}
```

**Corresponding InformationBlock:**

```c
typedef struct {
    EFI_STATUS MediaState;
} EFI_ADAPTER_INFO_MEDIA_STATE;
```

`MediaState` Returns the current media state status. `MediaState` can have any of the following values:

- **EFI_SUCCESS**: There is media attached to the network adapter.
**EFI_NOT_READY**: This detects a bounced state. There was media attached to the network adapter, but it was removed and is trying to attach to the network adapter again. If re-attached, the status will be updated to **EFI_SUCCESS** later.

**EFI_NO_MEDIA**: There is not any media attached to the network adapter.

### 11.12.2 Network Boot

For iSCSI and FCoE HBA adapters, the **EFI_ADAPTER_INFORMATION_PROTOCOL** must be installed on the same handle as the **EFI_EXT_SCSI_PASS_THRU_PROTOCOL**. When the **EFI_EXT_SCSI_PASS_THRU_PROTOCOL** cannot be installed because the adapter was not adequately configured, or if the relevant SCSI bus handles cannot be produced, this information must be installed on the controller handle that has been passed to the adapter Pass Thru Driver’s **EFI_DRIVER_BINDING_PROTOCOL.Start() function**. This will typically be a handle with the **EFI_PCI_IO_PROTOCOL** and **EFI_DEVICE_PATH_PROTOCOL**. If the handle with the **EFI_EXT_SCSI_PASS_THRU_PROTOCOL** is produced at a later time, the information on the controller handle must be uninstalled so as to avoid duplicate information.

**InformationType**

```c
#define EFI_ADAPTER_INFO_NETWORK_BOOT_GUID  
   {0x1FBD2960, 0x4130, 0x41E5,  
   {0x94,0xAC,0xD2, 0xCF, 0x03, 0x7F, 0xB3, 0x7C}}
```

**Corresponding InformationBlock**:

```c
typedef struct {
  BOOLEAN  iScsiIpv4BootCapablity;
  BOOLEAN  iScsiIpv6BootCapablity;
  BOOLEAN  FCoeBootCapablity;
  BOOLEAN  OffloadCapability;
  BOOLEAN  iScsiMpioCapability
  BOOLEAN  iScsiIpv4Boot;
  BOOLEAN  iScsiIpv6Boot;
  BOOLEAN  FCoeBoot;
} EFI_ADAPTER_INFO_NETWORK_BOOT;
```

- **iScsiIpv4BootCapablity**  
  TRUE if the adapter supports booting from iSCSI IPv4 targets.

- **iScsiIpv6BootCapablity**  
  TRUE if the adapter supports booting from iSCSI IPv6 targets.

- **FCoeBootCapablity**  
  TRUE if the adapter supports booting from FCoE targets.

- **OffloadCapability**  
  TRUE if the adapter supports an offload engine (such as TCP Offload Engine (TOE)) for its iSCSI or FCoE boot operations.

- **iScsiMpioCapability**
TRUE if the adapter supports multipath I/O (MPIO) for its iSCSI boot operations.

**iScsiIpv4Boot** TRUE if the adapter is currently configured to boot from iSCSI IPv4 targets.

**iScsiIpv6Boot** TRUE if the adapter is currently configured to boot from iSCSI IPv6 targets.

**FCoeBoot** TRUE if the adapter is currently configured to boot from FCoE targets.

**Note:** The adapter should set the **iScsiIpv4BootCapability**, **iScsiIpv6BootCapability**, or **FCoeBootCapability** fields to TRUE if it supports that boot capability, even if that capability is currently disabled or not configured.

### 11.12.3 SAN MAC Address

**SUMMARY**

This information block for the **EFI_ADAPTER_INFORMATION_PROTOCOL** supports ascertaining the SAN MAC address for an FCOE-aware network interface controller. This address is the Fabric-Provided MAC Address (FPMA) that gets assigned to the adapter port after the fabric login.

**Note:** An instance of the **EFI_ADAPTER_INFORMATION_PROTOCOL** supporting this GUID must be installed on the same handle as the **EFI_EXT_SCSI_PASS_THRU_PROTOCOL** when it is produced. However, this address is available to the adapter only when the fabric login has occurred, so in cases where the login cannot happen, where the adapter was not adequately configured, or if the relevant SCSI bus handles cannot be produced, this information type may not be produced.

**SAN MAC address information**

**InformationType**

```c
#define EFI_ADAPTER_INFO_SAN_MAC_ADDRESS_GUID \
{0x114da5ef, 0x2cf1, 0x4e12,\ 
  {0x9b, 0xbb, 0xc4, 0x70, 0xb5, 0x52, 0x05, 0xd9}}
```

**Corresponding InformationBlock:**

```c
typedef struct {
    EFI_MAC_ADDRESS SanMacAddress;
} EFI_ADAPTER_INFO_SAN_MAC_ADDRESS;
```

**SanMacAddress** Returns the SAN MAC address for the adapter.

### 11.12.4 IPV6 Support from UNDI

For network adapters, the **EFI_ADAPTER_INFORMATION_PROTOCOL** must be installed on the same handle as the UNDI protocol.

- **Ipv6Support** returns capability of UNDI to support IPV6 traffic.
- **Ipv6Support** can have any of the following values:
TRUE: The UNDI supports IPV6.
FALSE: This UNDI does not support IPV6 traffic.

Information Type

```
#define EFI_ADAPTER_INFO_UNDI_IPV6_SUPPORT_GUID
{ 0x4bd56be3, 0x4975, 0x4d8a, 
{0xa0, 0xad, 0xc4, 0x91, 0x20, 0x4b, 0x5d, 0x4d}}
```

Corresponding Information Block:

```
typedef struct {
    BOOLEAN IPv6Support;
} EFI_ADAPTER_INFO_UNDI_IPV6_SUPPORT;
```

11.12.5 Network Media Type

For network adapters, the EFI_ADAPTER_INFORMATION_PROTOCOL must be installed on the same handle as the UNDI protocol. If SNP or MNP protocol, instead of the UNDI protocol, is installed on adapter handle, then the EFI_ADAPTER_INFORMATION_PROTOCOL must be installed on the same handle as the SNP or MNP protocol.

Information Type

```
#define EFI_ADAPTER_INFO_MEDIA_TYPE_GUID
{ 0x8484472f, 0x71ec, 0x411a, 
{ 0xb3, 0x9c, 0x62, 0xcd, 0x94, 0xd9, 0x91, 0x6e }}
```

Corresponding Information Block:

```
typedef struct {
    UINT8 MediaType;
} EFI_ADAPTER_INFO_MEDIA_TYPE;
```

MediaType indicates the current media type, and can have any of the following values:

1: Ethernet Network Adapter
2: Ethernet Wireless Network Adapter
3~255: Reserved

11.12.6 Coherent Device Attribute Table (CDAT) Type

This section defines Adapter Information Protocol type for Coherent Device Attribute Table (CDAT). Compute Express Link (CXL) and other CPU-to-Memory interconnects enable coherent memory devices or coherent accelerator devices to be attached to the CPU. Unlike memory DIMMs, the system software or firmware may not have apriori knowledge of the attributes of memory located on these devices and would benefit from the device directly exposing the NUMA attributes such as latency and bandwidth characteristics. The necessary data structures are defined in the Coherent Device Attribute Table (CDAT)
structures. For more information, refer to “Links to UEFI-Related Documents” (http://uefi.org/uefi) under the heading "Coherent Device Attribute Table (CDAT) Specification".

**Note:** For CXL devices that support coherent memory, the `EFI_ADAPTER_INFORMATION_PROTOCOL` instance supporting this type may be installed by the EFI driver associated with this device on the device controller handle. This may happen during the driver initialization in the `EFI_IMAGE_ENTRY_POINT` of the driver, allowing for the CDAT structures to be published without relying on the Driver Model platform connect policy.

**Information Type**

```c
#define EFI_ADAPTER_INFO_CDAT_TYPE_GUID \
{0x77af24d1, 0xb6f0, 0x42b9, \
{0x83, 0xf5, 0x8f, 0xe6, 0xe8, 0x3e, 0xb6, 0xf0}}
```

**Corresponding InformationBlock:**

```c
typedef struct {
    UINTN CdatSize;
    UINT8 Cdat[];
} EFI_ADAPTER_INFO_CDAT_TYPE_TYPE;
```

*CdatSize* of the *Cdat* structure, in bytes.
*Cdat* Coherent Device Attribute Table (CDAT) structures.
12 - Protocols — Console Support

This section explores console support protocols, including Simple Text Input, Simple Text Output, Simple Pointer, Serial IO, and Graphics Output protocols.

12.1 Console I/O Protocol

This section defines the Console I/O protocol. This protocol is used to handle input and output of text-based information intended for the system user during the operation of code in the boot services environment. Also included here are the definitions of three console devices: one for input and one each for normal output and errors.

These interfaces are specified by function call definitions to allow maximum flexibility in implementation. For example, there is no requirement for compliant systems to have a keyboard or screen directly connected to the system. Implementations may choose to direct information passed using these interfaces in arbitrary ways provided that the semantics of the functions are preserved (in other words, provided that the information is passed to and from the system user).

12.1.1 Overview

The UEFI console is built out of the EFI_SIMPLE_TEXT_INPUT_PROTOCOL and the EFI_SIMPLE_TEXT_OUTPUT_PROTOCOL. These two protocols implement a basic text-based console that allows platform firmware, applications written to this specification, and UEFI OS loaders to present information to and receive input from a system administrator. The UEFI console supported 16-bit Unicode character codes, a simple set of input control characters (Scan Codes), and a set of output-oriented programmatic interfaces that give functionality equivalent to an intelligent terminal. The console does not support pointing devices on input or bitmaps on output.

This specification requires that the EFI_SIMPLE_TEXT_INPUT_PROTOCOL support the same languages as the corresponding EFI_SIMPLE_TEXT_OUTPUT_PROTOCOL. The EFI_SIMPLE_TEXT_OUTPUT_PROTOCOL is recommended to support at least the printable Basic Latin Unicode character set to enable standard terminal emulation software to be used with an EFI console. The Basic Latin Unicode character set implements a superset of ASCII that has been extended to 16-bit characters. Any number of other Unicode character sets may be optionally supported.

12.1.2 ConsoleIn Definition

The EFI_SIMPLE_TEXT_INPUT_PROTOCOL defines an input stream that contains Unicode characters and required EFI scan codes. Only the control characters defined in Table 12-1 have meaning in the Unicode input or output streams. The control characters are defined to be characters U+0000 through U+001F. The input stream does not support any software flow control.

<table>
<thead>
<tr>
<th>Mnemonic</th>
<th>Unicode</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Null</td>
<td>U+0000</td>
<td>Null character ignored when received.</td>
</tr>
<tr>
<td>BS</td>
<td>U+0008</td>
<td>Backspace. Moves cursor left one column. If the cursor is at the left margin, no action is taken.</td>
</tr>
<tr>
<td>TAB</td>
<td>U+0x0009</td>
<td>Tab.</td>
</tr>
</tbody>
</table>
The input stream supports Scan Codes in addition to Unicode characters. If the Scan Code is set to 0x00 then the Unicode character is valid and should be used. If the Scan Code is set to a non-0x00 value it represents a special key as defined by Table 12-2.

**Table 12-2 EFI Scan Codes for EFI_SIMPLE_TEXT_INPUT_PROTOCOL**

<table>
<thead>
<tr>
<th>EFI Scan Code</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x00</td>
<td>Null scan code.</td>
</tr>
<tr>
<td>0x01</td>
<td>Move cursor up 1 row.</td>
</tr>
<tr>
<td>0x02</td>
<td>Move cursor down 1 row.</td>
</tr>
<tr>
<td>0x03</td>
<td>Move cursor right 1 column.</td>
</tr>
<tr>
<td>0x04</td>
<td>Move cursor left 1 column.</td>
</tr>
<tr>
<td>0x05</td>
<td>Home.</td>
</tr>
<tr>
<td>0x06</td>
<td>End.</td>
</tr>
<tr>
<td>0x07</td>
<td>Insert.</td>
</tr>
<tr>
<td>0x08</td>
<td>Delete.</td>
</tr>
<tr>
<td>0x09</td>
<td>Page Up.</td>
</tr>
<tr>
<td>0x0a</td>
<td>Page Down.</td>
</tr>
<tr>
<td>0x0b</td>
<td>Function 1.</td>
</tr>
<tr>
<td>0x0c</td>
<td>Function 2.</td>
</tr>
<tr>
<td>0x0d</td>
<td>Function 3.</td>
</tr>
<tr>
<td>0x0e</td>
<td>Function 4.</td>
</tr>
<tr>
<td>0x0f</td>
<td>Function 5.</td>
</tr>
<tr>
<td>0x10</td>
<td>Function 6.</td>
</tr>
<tr>
<td>0x11</td>
<td>Function 7.</td>
</tr>
<tr>
<td>0x12</td>
<td>Function 8.</td>
</tr>
<tr>
<td>0x13</td>
<td>Function 9.</td>
</tr>
<tr>
<td>0x14</td>
<td>Function 10.</td>
</tr>
<tr>
<td>0x17</td>
<td>Escape.</td>
</tr>
</tbody>
</table>

**Table 12-3EFI Scan Codes for EFI_SIMPLE_TEXT_INPUT_EX_PROTOCOL**

<table>
<thead>
<tr>
<th>EFI Scan Code</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x15</td>
<td>Function 11</td>
</tr>
<tr>
<td>0x16</td>
<td>Function 12</td>
</tr>
</tbody>
</table>
The Simple Text Input Ex protocol defines an extension to the Simple Text Input protocol which enables various new capabilities described in this section.

<table>
<thead>
<tr>
<th>EFI Scan Code</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x68</td>
<td>Function 13</td>
</tr>
<tr>
<td>0x69</td>
<td>Function 14</td>
</tr>
<tr>
<td>0x6A</td>
<td>Function 15</td>
</tr>
<tr>
<td>0x6B</td>
<td>Function 16</td>
</tr>
<tr>
<td>0x6C</td>
<td>Function 17</td>
</tr>
<tr>
<td>0x6D</td>
<td>Function 18</td>
</tr>
<tr>
<td>0x6E</td>
<td>Function 19</td>
</tr>
<tr>
<td>0x6F</td>
<td>Function 20</td>
</tr>
<tr>
<td>0x70</td>
<td>Function 21</td>
</tr>
<tr>
<td>0x71</td>
<td>Function 22</td>
</tr>
<tr>
<td>0x72</td>
<td>Function 23</td>
</tr>
<tr>
<td>0x73</td>
<td>Function 24</td>
</tr>
<tr>
<td>0x7F</td>
<td>Mute</td>
</tr>
<tr>
<td>0x80</td>
<td>Volume Up</td>
</tr>
<tr>
<td>0x81</td>
<td>Volume Down</td>
</tr>
<tr>
<td>0x100</td>
<td>Brightness Up</td>
</tr>
<tr>
<td>0x101</td>
<td>Brightness Down</td>
</tr>
<tr>
<td>0x102</td>
<td>Suspend</td>
</tr>
<tr>
<td>0x103</td>
<td>Hibernate</td>
</tr>
<tr>
<td>0x104</td>
<td>Toggle Display</td>
</tr>
<tr>
<td>0x105</td>
<td>Recovery</td>
</tr>
<tr>
<td>0x106</td>
<td>Eject</td>
</tr>
<tr>
<td>0x8000-0xFFFF</td>
<td>OEM Reserved</td>
</tr>
</tbody>
</table>
EFI_SIMPLE_TEXT_INPUT_EX_PROTOCOL

Summary
This protocol is used to obtain input from the ConsoleIn device. The EFI specification requires that the EFI_SIMPLE_TEXT_INPUT_PROTOCOL supports the same languages as the corresponding EFI_SIMPLE_TEXT_OUTPUT_PROTOCOL.

GUID
#define EFI_SIMPLE_TEXT_INPUT_EX_PROTOCOL_GUID \
{0xdd9e7534, 0x7762, 0x4698, \
{0x8c, 0x14, 0xf5, 0x85, 0x17, 0xa6, 0x25, 0xaa}}

Protocol Interface Structure
typedef struct _EFI_SIMPLE_TEXT_INPUT_EX_PROTOCOL{
  EFI_INPUT_RESET_EX Reset;
  EFI_INPUT_READ_KEY_EX ReadKeyStrokeEx;
  EFI_EVENT WaitForKeyEx;
  EFI_SET_STATE SetState;
  EFI_REGISTER_KEYSTROKE_NOTIFY RegisterKeyNotify;
  EFI_UNREGISTER_KEYSTROKE_NOTIFY UnregisterKeyNotify;
} EFI_SIMPLE_TEXT_INPUT_EX_PROTOCOL;

Parameters
Reset Reset the ConsoleIn device. See Reset().
ReadKeyStrokeEx Returns the next input character. See ReadKeyStrokeEx().
WaitForKeyEx Event to use with WaitForEvent() to wait for a key to be available. An Event will only be triggered if KeyData.Key has information contained within it.
SetState Set the EFI_KEY_TOGGLE_STATE state settings for the input device.
RegisterKeyNotify Register a notification function to be called when a given key sequence is hit.
UnregisterKeyNotify Removes a specific notification function.

Description
The EFI_SIMPLE_TEXT_INPUT_EX_PROTOCOL is used on the ConsoleIn device. It is an extension to the Simple Text Input protocol which allows a variety of extended shift state information to be returned.

EFI_SIMPLE_TEXT_INPUT_EX_PROTOCOL.Reset()

Summary
Resets the input device hardware.
Prototype

```c
typedef EFI_STATUS
  (EFIAPIC *EFI_INPUT_RESET_EX) (
    IN EFI_SIMPLE_TEXT_INPUT_EX_PROTOCOL *This,
    IN BOOLEAN ExtendedVerification
  );
```

Parameters

- **This**: A pointer to the `EFI_SIMPLE_TEXT_INPUT_EX_PROTOCOL` instance. Type `EFI_SIMPLE_TEXT_INPUT_EX_PROTOCOL` is defined in this section.
- **ExtendedVerification**: Indicates that the driver may perform a more exhaustive verification operation of the device during reset.

Description

The `Reset()` function resets the input device hardware.

The implementation of `Reset` is required to clear the contents of any input queues resident in memory used for buffering keystroke data and put the input stream in a known empty state.

As part of initialization process, the firmware/device will make a quick but reasonable attempt to verify that the device is functioning. If the `ExtendedVerification` flag is `TRUE` the firmware may take an extended amount of time to verify the device is operating on reset. Otherwise the reset operation is to occur as quickly as possible.

The hardware verification process is not defined by this specification and is left up to the platform firmware or driver to implement.

Status Codes Returned

<table>
<thead>
<tr>
<th>Status Code</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>EFI_SUCCESS</td>
<td>The device was reset.</td>
</tr>
<tr>
<td>EFI_DEVICE_ERROR</td>
<td>The device is not functioning correctly and could not be reset.</td>
</tr>
</tbody>
</table>

**EFI_SIMPLE_TEXT_INPUT_EX_PROTOCOL.ReadKeyStrokeEx()**

**Summary**

Reads the next keystroke from the input device.
Prototype
typedef
EFI_STATUS
(EIFIAPI *EFI_INPUT_READ_KEY_EX) (  
    IN EFI_SIMPLE_TEXT_INPUT_EX_PROTOCOL *This,
    OUT EFI_KEY_DATA *KeyData
  );

Parameters
This
A pointer to the EFI_SIMPLE_TEXT_INPUT_EX_PROTOCOL instance. Type EFI_SIMPLE_TEXT_INPUT_EX_PROTOCOL is defined in this section.

KeyData
A pointer to a buffer that is filled in with the keystroke state data for the key that was pressed. Type EFI_KEY_DATA is defined in "Related Definitions" below.

Related Definitions
//*******************************************************
// EFI_KEY_DATA
//*******************************************************
typedef struct {
    EFI_INPUT_KEY Key;
    EFI_KEY_STATE KeyState;
} EFI_KEY_DATA

Key
The EFI scan code and Unicode value returned from the input device.

KeyState
The current state of various toggled attributes as well as input modifier values.

//*******************************************************
// EFI_KEY_STATE
//*******************************************************
//
// Any Shift or Toggle State that is valid should have
// high order bit set.
//
typedef struct EFI_KEY_STATE {
    UINT32 KeyShiftState;
    EFI_KEY_TOGGLE_STATE KeyToggleState;
} EFI_KEY_STATE;

KeyShiftState
Reflects the currently pressed shift modifiers for the input device. The returned value is valid only if the high order bit has been set.

KeyToggleState
Reflects the current internal state of various toggled attributes. The returned value is valid only if the high order bit has been set.
#define EFI_SHIFT_STATE_VALID 0x80000000
#define EFI_RIGHT_SHIFT_PRESSED 0x00000001
#define EFI_LEFT_SHIFT_PRESSED 0x00000002
#define EFI_RIGHT_CONTROL_PRESSED 0x00000004
#define EFI_LEFT_CONTROL_PRESSED 0x00000008
#define EFI_RIGHT_ALT_PRESSED 0x00000010
#define EFI_LEFT_ALT_PRESSED 0x00000020
#define EFI_RIGHT_LOGO_PRESSED 0x00000040
#define EFI_LEFT_LOGO_PRESSED 0x00000080
#define EFI_MENU_KEY_PRESSED 0x00000100
#define EFI_SYS_REQ_PRESSED 0x00000200

//*******************************************************
// EFI_KEY_TOGGLE_STATE
//**********************************************************
typedef UINT8 EFI_KEY_TOGGLE_STATE;

#define EFI_TOGGLE_STATE_VALID 0x80
#define EFI_KEY_STATE_EXPOSED 0x40
#define EFI_SCROLL_LOCK_ACTIVE 0x01
#define EFI_NUM_LOCK_ACTIVE 0x02
#define EFI_CAPS_LOCK_ACTIVE 0x04

Description

The `ReadKeyStrokeEx()` function reads the next keystroke from the input device. If there is no pending keystroke the function returns `EFI_NOT_READY`. If there is a pending keystroke, then `KeyData.Key.ScanCode` is the EFI scan code defined in Table 12-2. The `KeyData.Key.UnicodeChar` is the actual printable character or is zero if the key does not represent a printable character (control key, function key, etc.). The `KeyData.KeyState` is the modifier shift state for the character reflected in `KeyData.Key.UnicodeChar` or `KeyData.Key.ScanCode`. This function mirrors the behavior of `ReadKeyStroke()` in the Simple Input Protocol in that a keystroke will only be returned when `KeyData.Key` has data within it.

When interpreting the data from this function, it should be noted that if a class of printable characters that are normally adjusted by shift modifiers (e.g. Shift Key + "f" key) would be presented solely as a `KeyData.Key.UnicodeChar` without the associated shift state. So in the previous example of a Shift Key + "f" key being pressed, the only pertinent data returned would be `KeyData.Key.UnicodeChar` with the value of "F". This of course would not typically be the case for non-printable characters such as the pressing of the Right Shift Key + F10 key since the corresponding returned data would be reflected both in the `KeyData.KeyState.KeyShiftState` and `KeyData.Key.ScanCode` values.

UEFI drivers which implement the `EFI_SIMPLE_TEXT_INPUT_EX` protocol are required to return `KeyData.Key` and `KeyData.KeyState` values. These drivers must always return the most current state of `KeyData.KeyState.KeyShiftState` and `KeyData.KeyState.KeyToggleState`. It should also be noted that certain input devices may not be able to produce shift or toggle state information, and in those cases the high order bit in the respective Toggle and Shift state fields should not be active.
If the `EFI_KEY_STATE_EXPOSED` bit is turned on, then this instance of the `EFI_SIMPLE_INPUT_EX_PROTOCOL` supports the ability to return partial keystrokes. With `EFI_KEY_STATE_EXPOSED` bit enabled, the `ReadKeyStrokeEx` function will allow the return of incomplete keystrokes such as the holding down of certain keys which are expressed as a part of `KeyState` when there is no `Key` data.

### Status Codes Returned

<table>
<thead>
<tr>
<th>Code</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>EFI_SUCCESS</td>
<td>The keystroke information was returned.</td>
</tr>
<tr>
<td>EFI_NOT_READY</td>
<td>There was no keystroke data available.. Current KeyData.KeyState values are exposed.</td>
</tr>
<tr>
<td>EFIDEVICEERROR</td>
<td>The keystroke information was not returned due to hardware errors.</td>
</tr>
<tr>
<td>EFI_UNSUPPORTED</td>
<td>The device does not support the ability to read keystroke data.</td>
</tr>
</tbody>
</table>

### EFI_SIMPLE_TEXT_INPUT_EX_PROTOCOL.SetState()

**Summary**

Set certain state for the input device.

**Prototype**

```c
typedef EFI_STATUS (EFIAPI *EFI_SET_STATE) (    
    IN EFI_SIMPLE_TEXT_INPUT_EX_PROTOCOL *This,    
    IN EFI_KEY_TOGGLE_STATE *KeyToggleState    
);
```

**Parameters**

- **This**: A pointer to the `EFI_SIMPLE_TEXT_INPUT_EX_PROTOCOL` instance. Type `EFI_SIMPLE_TEXT_INPUT_EX_PROTOCOL` is defined in this section.
- **KeyToggleState**: Pointer to the `EFI_KEY_TOGGLE_STATE` to set the state for the input device. Type `EFI_KEY_TOGGLE_STATE` is defined in "Related Definitions" for `EFI_SIMPLE_TEXT_INPUT_EX_PROTOCOL.ReadKeyStrokeEx()`, above.

The `SetState()` function allows the input device hardware to have state settings adjusted. By calling the `SetState()` function with the `EFI_KEY_STATE_EXPOSED` bit active in the `KeyToggleState` parameter, this will enable the `ReadKeyStrokeEx` function to return incomplete keystrokes such as the holding down of certain keys which are expressed as a part of `KeyState` when there is no `Key` data.
Status Codes Returned

<table>
<thead>
<tr>
<th>Status Code</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>EFI_SUCCESS</td>
<td>The device state was set appropriately.</td>
</tr>
<tr>
<td>EFI_DEVICE_ERROR</td>
<td>The device is not functioning correctly and could not have the setting adjusted.</td>
</tr>
<tr>
<td>EFI_UNSUPPORTED</td>
<td>The device does not support the ability to have its state set or the requested state change was not supported.</td>
</tr>
</tbody>
</table>

**EFI_SIMPLE_TEXT_INPUT_EX_PROTOCOL.RegisterKeyNotify()**

**Summary**
Register a notification function for a particular keystroke for the input device.

**Prototype**
```c
typedef EFI_STATUS (EFIAPI *EFI_REGISTER_KEYSTROKE_NOTIFY) (  
    IN EFI_SIMPLE_TEXT_INPUT_EX_PROTOCOL *This,  
    IN EFI_KEY_DATA                      *KeyData,  
    IN EFI_KEY_NOTIFY_FUNCTION            KeyNotificationFunction,  
    OUT VOID                             **NotifyHandle
);
```

**Parameters**
- **This**
  A pointer to the `EFI_SIMPLE_TEXT_INPUT_EX_PROTOCOL` instance. Type `EFI_SIMPLE_TEXT_INPUT_EX_PROTOCOL` is defined in this section.
- **KeyData**
  A pointer to a buffer that is filled in with the keystroke information for the key that was pressed. If KeyData.Key, KeyData.KeyState.KeyToggleState and KeyData.KeyState.KeyShiftState are 0, then any incomplete keystroke will trigger a notification of the KeyNotificationFunction.
- **KeyNotificationFunction**
  Points to the function to be called when the key sequence is typed specified by KeyData. This notification function should be called at `<=TPL_CALLBACK`. See `EFI_KEY_NOTIFY_FUNCTION` below.
- **NotifyHandle**
  Points to the unique handle assigned to the registered notification.

**Description**

The `RegisterKeystrokeNotify()` function registers a function which will be called when a specified keystroke will occur. The keystroke being specified can be for any combination of `KeyData.Key` or `KeyData.KeyState` information.
Related Definitions

```c
typedef EFI_STATUS
(EFI_API *EFI_KEY_NOTIFY_FUNCTION) (IN EFI_KEY_DATA *KeyData);
```

Status Codes Returned

<table>
<thead>
<tr>
<th>Code</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>EFI_SUCCESS</td>
<td>Key notify was registered successfully.</td>
</tr>
<tr>
<td>EFI_OUT_OF_RESOURCES</td>
<td>Unable to allocate necessary data structures.</td>
</tr>
</tbody>
</table>

**EFI_SIMPLE_TEXT_INPUT_EX_PROTOCOL.UnregisterKeyNotify()**

**Summary**
Remove the notification that was previously registered.

**Prototype**

```c
typedef EFI_STATUS
(EFI_API *EFI_UNREGISTER_KEYSTROKE_NOTIFY) (IN EFI_SIMPLE_TEXT_INPUT_EX_PROTOCOL *This,
IN VOID *NotificationHandle);
```

**Parameters**

- **This**
  A pointer to the `EFI_SIMPLE_TEXT_INPUT_EX_PROTOCOL` instance. Type `EFI_SIMPLE_TEXT_INPUT_EX_PROTOCOL` is defined in this section.
- **NotificationHandle**
  The handle of the notification function being unregistered.

**Description**

The `UnregisterKeystrokeNotify()` function removes the notification which was previously registered.

**Status Codes Returned**

<table>
<thead>
<tr>
<th>Code</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>EFI_SUCCESS</td>
<td>Key notify was unregistered successfully.</td>
</tr>
<tr>
<td>EFI_INVALID_PARAMETER</td>
<td>The NotificationHandle is invalid.</td>
</tr>
</tbody>
</table>

**12.3 Simple Text Input Protocol**

The Simple Text Input protocol defines the minimum input required to support the `ConsoleIn` device.
EFI_SIMPLE_TEXT_INPUT_PROTOCOL

Summary
This protocol is used to obtain input from the ConsoleIn device. The EFI specification requires that the EFI_SIMPLE_TEXT_INPUT_PROTOCOL supports the same languages as the corresponding EFI_SIMPLE_TEXT_OUTPUT_PROTOCOL.

GUID
#define EFI_SIMPLE_TEXT_INPUT_PROTOCOL_GUID \
{0x387477c1,0x69c7,0x11d2,\ 
{0x8e,0x39,0x00,0xa0,0xc9,0x72,0x3b}}

Protocol Interface Structure
typedef struct _EFI_SIMPLE_TEXT_INPUT_PROTOCOL { 
    EFI_INPUT_RESET    Reset; 
    EFI_INPUT_READ_KEY ReadKeyStroke; 
    EFI_EVENT         WaitForKey; 
} EFI_SIMPLE_TEXT_INPUT_PROTOCOL;

Parameters
Reset                      Reset the ConsoleIn device. See Reset().
_ReadKeyStroke_            Returns the next input character. See ReadKeyStroke().
WaitForKey                 Event to use with EFI_BOOT_SERVICES.WaitForEvent() to wait for a key to be available.

Description
The EFI_SIMPLE_TEXT_INPUT_PROTOCOL is used on the ConsoleIn device. It is the minimum required protocol for ConsoleIn.

EFI_SIMPLE_TEXT_INPUT_PROTOCOL.Reset()

Summary
Resets the input device hardware.
Prototype

typedef

EFI_STATUS

(EIFIAP  *EFI_INPUT_RESET) (  

IN  EFI_SIMPLE_TEXT_INPUT_PROTOCOL  *This,  

IN BOOLEAN  ExtendedVerification

);

Parameters

This

A pointer to the EFI_SIMPLE_TEXT_INPUT_PROTOCOL instance. Type EFI_SIMPLE_TEXT_INPUT_PROTOCOL is defined in Section 12.3

ExtendedVerification

Indicates that the driver may perform a more exhaustive verification operation of the device during reset.

Description

The Reset() function resets the input device hardware.

The implementation of Reset is required to clear the contents of any input queues resident in memory used for buffering keystroke data and put the input stream in a known empty state.

As part of initialization process, the firmware/device will make a quick but reasonable attempt to verify that the device is functioning. If the ExtendedVerification flag is TRUE the firmware may take an extended amount of time to verify the device is operating on reset. Otherwise the reset operation is to occur as quickly as possible.

The hardware verification process is not defined by this specification and is left up to the platform firmware or driver to implement.

Status Codes Returned

<table>
<thead>
<tr>
<th>Code</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>EFI_SUCCESS</td>
<td>The device was reset.</td>
</tr>
<tr>
<td>EFI_DEVICE_ERROR</td>
<td>The device is not functioning correctly and could not be reset.</td>
</tr>
</tbody>
</table>

EFI_SIMPLE_TEXT_INPUT_PROTOCOL.ReadKeyStroke()

Summary

Reads the next keystroke from the input device.
Prototype

typedef
    EFI_STATUS
    (EFIAPI *EFI_INPUT_READ_KEY) (    
    IN EFI_SIMPLE_TEXT_INPUT_PROTOCOL *This,
    OUT EFI_INPUT_KEY *Key
    );

Parameters

This   A pointer to the EFI_SIMPLE_TEXT_INPUT_PROTOCOL instance. Type EFI_SIMPLE_TEXT_INPUT_PROTOCOL is defined in Section 12.3.

Key    A pointer to a buffer that is filled in with the keystroke information for the key that was pressed. Type EFI_INPUT_KEY is defined in “Related Definitions” below.

Related Definitions

    //*******************************************************************************
    // EFI_INPUT_KEY
    //*******************************************************************************
    typedef struct {
        UINT16      ScanCode;
        CHAR16     UnicodeChar;
    } EFI_INPUT_KEY;

Description

The ReadKeyStroke() function reads the next keystroke from the input device. If there is no pending keystroke the function returns EFI_NOT_READY. If there is a pending keystroke, then ScanCode is the EFI scan code defined in Table 12-2. The UnicodeChar is the actual printable character or is zero if the key does not represent a printable character (control key, function key, etc.).

Status Codes Returned

<table>
<thead>
<tr>
<th>Status Code</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>EFI_SUCCESS</td>
<td>The keystroke information was returned.</td>
</tr>
<tr>
<td>EFI_NOT_READY</td>
<td>There was no keystroke data available.</td>
</tr>
<tr>
<td>EFI_DEVICE_ERROR</td>
<td>The keystroke information was not returned due to hardware errors.</td>
</tr>
<tr>
<td>EFI_UNSUPPORTED</td>
<td>The device does not support the ability to read keystroke data.</td>
</tr>
</tbody>
</table>

12.3.1 ConsoleOut or StandardError

The EFI_SIMPLE_TEXT_OUTPUT_PROTOCOL must implement the same Unicode code pages as the EFI_SIMLPLE_TEXT_INPUT_PROTOCOL. The protocol must also support the Unicode control characters defined in Table 12-1. The EFI_SIMPLE_TEXT_OUTPUT_PROTOCOL supports special manipulation of the screen by programmatic methods and therefore does not support the EFI scan codes defined in Table 12-2.
12.4 Simple Text Output Protocol

The Simple Text Output protocol defines the minimum requirements for a text-based ConsoleOut device. The EFI specification requires that the EFI_SIMPLE_TEXT_INPUT_PROTOCOL support the same languages as the corresponding EFI_SIMPLE_TEXT_OUTPUT_PROTOCOL.

EFI_SIMPLE_TEXT_OUTPUT_PROTOCOL

Summary

This protocol is used to control text-based output devices.

GUID

```c
#define EFI_SIMPLE_TEXT_OUTPUT_PROTOCOL_GUID \
{0x387477c2,0x69c7,0x11d2,\ 
{0x8e,0x39,0xa0,0xc9,0x72,0x3b}}
```

Protocol Interface Structure

```c
typedef struct _EFI_SIMPLE_TEXT_OUTPUT_PROTOCOL {  
  EFI_TEXT_RESET          Reset;  
  EFI_TEXT_STRING         OutputString;  
  EFI_TEXT_TEST_STRING    TestString;  
  EFI_TEXT_QUERY_MODE    QueryMode;  
  EFI_TEXT_SET_MODE      SetMode;  
  EFI_TEXT_SET_ATTRIBUTE SetAttribute;  
  EFI_TEXT_CLEAR_SCREEN  ClearScreen;  
  EFI_TEXT_SET_CURSOR_POSITION SetCursorPosition;  
  EFI_TEXT_ENABLE_CURSOR EnableCursor;  
  SIMPLE_TEXT_OUTPUT_MODE *Mode;
} EFI_SIMPLE_TEXT_OUTPUT_PROTOCOL;
```

Parameters

- **Reset**
  Reset the ConsoleOut device. See [Reset()](#).
- **OutputString**
  Displays the string on the device at the current cursor location. See [OutputString()](#).
- **TestString**
  Tests to see if the ConsoleOut device supports this string. See [TestString()](#).
- **QueryMode**
  Queries information concerning the output device’s supported text mode. See [QueryMode()](#).
- **SetMode**
  Sets the current mode of the output device. See [SetMode()](#).
- **setAttribute**
  Sets the foreground and background color of the text that is output. See [SetAttribute()](#).
- **ClearScreen**
  Clears the screen with the currently set background color. See [ClearScreen()](#).
- **SetCursorPosition**
  Sets the current cursor position. See [SetCursorPosition()](#).
- **EnableCursor**
  Turns the visibility of the cursor on/off. See [EnableCursor()](#).
Mode

Pointer to `SIMPLE_TEXT_OUTPUT_MODE` data. Type `SIMPLE_TEXT_OUTPUT_MODE` is defined in “Related Definitions” below.

The following data values in the `SIMPLE_TEXT_OUTPUT_MODE` interface are read-only and are changed by using the appropriate interface functions:

- **MaxMode**: The number of modes supported by `QueryMode()` and `SetMode()`.
- **Mode**: The text mode of the output device(s).
- **Attribute**: The current character output attribute.
- **CursorPosition**: The cursor’s column.
- **CursorRow**: The cursor’s row.
- **CursorVisible**: The cursor is currently visible or not.

### Related Definitions

```c
//*******************************************************
// SIMPLE_TEXT_OUTPUT_MODE
//*******************************************************
typedef struct {
    INT32    MaxMode;
    // current settings
    INT32    Mode;
    INT32    Attribute;
    INT32    CursorColumn;
    INT32    CursorRow;
    BOOLEAN  CursorVisible;
} SIMPLE_TEXT_OUTPUT_MODE;
```

### Description

The `SIMPLE_TEXT_OUTPUT` protocol is used to control text-based output devices. It is the minimum required protocol for any handle supplied as the `ConsoleOut` or `StandardError` device. In addition, the minimum supported text mode of such devices is at least 80 x 25 characters.

A video device that only supports graphics mode is required to emulate text mode functionality. Output strings themselves are not allowed to contain any control codes other than those defined in Table 12-1. Positional cursor placement is done only via the `SetCursorPosition()` function. It is highly recommended that text output to the `StandardError` device be limited to sequential string outputs. (That is, it is not recommended to use `ClearScreen()` or `SetCursorPosition()` on output messages to `StandardError`.)

If the output device is not in a valid text mode at the time of the `EFI_BOOT_SERVICES.HandleProtocol()` call, the device is to indicate that its `CurrentMode` is −1. On connecting to the output device the caller is required to verify the mode of the output device, and if it is not acceptable to set it to something it can use.
EFI_SIMPLE_TEXT_OUTPUT_PROTOCOL.Reset()

Summary
Resets the text output device hardware.

Prototype

typedef
   EFI_STATUS
(EFI_API *EFI_TEXT_RESET) (   
   IN EFI_SIMPLE_TEXT_OUTPUT_PROTOCOL *This,
   IN BOOLEAN ExtendedVerification
);

Parameters

   This A pointer to the EFI_SIMPLE_TEXT_OUTPUT_PROTOCOL instance. Type EFI_SIMPLE_TEXT_OUTPUT_PROTOCOL is defined in the “Related Definitions” of Section 12.4.

   ExtendedVerification Indicates that the driver may perform a more exhaustive verification operation of the device during reset.

Description

The Reset() function resets the text output device hardware. The cursor position is set to (0, 0), and the screen is cleared to the default background color for the output device.

As part of initialization process, the firmware/device will make a quick but reasonable attempt to verify that the device is functioning. If the ExtendedVerification flag is TRUE the firmware may take an extended amount of time to verify the device is operating on reset. Otherwise the reset operation is to occur as quickly as possible.

The hardware verification process is not defined by this specification and is left up to the platform firmware or driver to implement.

Status Codes Returned

<table>
<thead>
<tr>
<th>Code</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>EFI_SUCCESS</td>
<td>The text output device was reset.</td>
</tr>
<tr>
<td>EFI_DEVICE_ERROR</td>
<td>The text output device is not functioning correctly and could not be reset.</td>
</tr>
</tbody>
</table>

EFI_SIMPLE_TEXT_OUTPUT_PROTOCOL.OutputString()

Summary
Writes a string to the output device.
Prototype

typedef
    EFI_STATUS
    (EFIAPI *EFI_TEXT_STRING) (  
    IN EFI_SIMPLE_TEXT_OUTPUT_PROTOCOL *This,
    IN CHAR16 *String
    );

Parameters

This

A pointer to the EFI_SIMPLE_TEXT_OUTPUT_PROTOCOL instance. Type EFI_SIMPLE_TEXT_OUTPUT_PROTOCOL is defined in the “Related Definitions” of Section 12.4.

String

The Null-terminated string to be displayed on the output device(s). All output devices must also support the Unicode drawing character codes defined in “Related Definitions.”

Related Definitions

//*******************************************************
// UNICODE DRAWING CHARACTERS
//*******************************************************
#define BOXDRAW_HORIZONTAL               0x2500
#define BOXDRAW_VERTICAL                 0x2502
#define BOXDRAW_DOWN_RIGHT               0x250c
#define BOXDRAW_DOWN_LEFT                0x2510
#define BOXDRAW_UP_RIGHT                 0x2514
#define BOXDRAW_UP_LEFT                  0x2518
#define BOXDRAW_VERTICAL_RIGHT           0x251c
#define BOXDRAW_VERTICAL_LEFT            0x2524
#define BOXDRAW_DOWN_HORIZONTAL          0x252c
#define BOXDRAW_UP_HORIZONTAL            0x2534
#define BOXDRAW_VERTICAL_HORIZONTAL      0x253c
#define BOXDRAW_DOUBLE_HORIZONTAL        0x2550
#define BOXDRAW_DOUBLE_VERTICAL          0x2551
#define BOXDRAW_DOWN_RIGHT_DOUBLE        0x2552
#define BOXDRAW_DOWN_DOUBLE_RIGHT        0x2553
#define BOXDRAW_DOUBLE_DOWN_RIGHT        0x2554
#define BOXDRAW_DOWN_LEFT_DOUBLE         0x2555
#define BOXDRAW_DOWN_DOUBLE_LEFT         0x2556
#define BOXDRAW_DOUBLE_DOWN_LEFT         0x2557
#define BOXDRAW_UP_RIGHT_DOUBLE          0x2558
#define BOXDRAW_UP_DOUBLE_RIGHT          0x2559
#define BOXDRAW_DOUBLE_UP_RIGHT          0x255a
```c
#define BOXDRAW_UP_LEFT_DOUBLE 0x255b
#define BOXDRAW_UP_DOUBLE_LEFT 0x255c
#define BOXDRAW_DOUBLE_UP_LEFT 0x255d
#define BOXDRAW_VERTICAL_RIGHT_DOUBLE 0x255e
#define BOXDRAW_VERTICAL_DOUBLE_RIGHT 0x255f
#define BOXDRAW_DOUBLE_VERTICAL_RIGHT 0x2560
#define BOXDRAW_VERTICAL_LEFT_DOUBLE 0x2561
#define BOXDRAW_VERTICAL_DOUBLE_LEFT 0x2562
#define BOXDRAW_DOUBLE_VERTICAL_LEFT 0x2563
#define BOXDRAW_DOWN_HORIZONTAL_DOUBLE 0x2564
#define BOXDRAW_DOWN_DOUBLE_HORIZONTAL 0x2565
#define BOXDRAW_DOUBLE_DOWN_HORIZONTAL 0x2566
#define BOXDRAW_UP_HORIZONTAL_DOUBLE 0x2567
#define BOXDRAW_UP_DOUBLE_HORIZONTAL 0x2568
#define BOXDRAW_DOUBLE_UP_HORIZONTAL 0x2569
#define BOXDRAW_VERTICAL_HORIZONTAL_DOUBLE 0x256a
#define BOXDRAW_VERTICAL_DOUBLE_HORIZONTAL 0x256b
#define BOXDRAW_DOUBLE_VERTICAL_HORIZONTAL 0x256c

/****************************************************************************
 /* EFI Required Block Elements Code Chart
/****************************************************************************

#define BLOCKELEMENT_FULL_BLOCK 0x2588
#define BLOCKELEMENT_LIGHT_SHADE 0x2591

/****************************************************************************
 /* EFI Required Geometric Shapes Code Chart
/****************************************************************************

#define GEOMETRICSHAPE_UP_TRIANGLE 0x25b2
#define GEOMETRICSHAPE_RIGHT_TRIANGLE 0x25ba
#define GEOMETRICSHAPE_DOWN_TRIANGLE 0x25bc
#define GEOMETRICSHAPE_LEFT_TRIANGLE 0x25c4

/****************************************************************************
 /* EFI Required Arrow shapes
/****************************************************************************

#define ARROW_UP 0x2191
#define ARROW_DOWN 0x2193
```
Description
The `OutputString()` function writes a string to the output device. This is the most basic output mechanism on an output device. The `String` is displayed at the current cursor location on the output device(s) and the cursor is advanced according to the rules listed in Table 12-4.

Table 12-4 EFI Cursor Location/Advance Rules

<table>
<thead>
<tr>
<th>Mnemonic</th>
<th>Unicode</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Null</td>
<td>U+0000</td>
<td>Ignore the character, and do not move the cursor.</td>
</tr>
<tr>
<td>BS</td>
<td>U+0008</td>
<td>If the cursor is not at the left edge of the display, then move the cursor left one column.</td>
</tr>
<tr>
<td>LF</td>
<td>U+000A</td>
<td>If the cursor is at the bottom of the display, then scroll the display one row, and do not update the cursor position. Otherwise, move the cursor down one row.</td>
</tr>
<tr>
<td>CR</td>
<td>U+000D</td>
<td>Move the cursor to the beginning of the current row.</td>
</tr>
<tr>
<td>Other</td>
<td>U+XXXX</td>
<td>Print the character at the current cursor position and move the cursor right one column. If this moves the cursor past the right edge of the display, then the line should wrap to the beginning of the next line. This is equivalent to inserting a CR and an LF. Note that if the cursor is at the bottom of the display, and the line wraps, then the display will be scrolled one line.</td>
</tr>
</tbody>
</table>

Note: If desired, the system’s NVRAM environment variables may be used at install time to determine the configured locale of the system or the installation procedure can query the user for the proper language support. This is then used to either install the proper EFI image/loader or to configure the installed image’s strings to use the proper text for the selected locale.

Status Codes Returned

<table>
<thead>
<tr>
<th>Code</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>EFI_SUCCESS</td>
<td>The string was output to the device.</td>
</tr>
<tr>
<td>EFI_DEVICE_ERROR</td>
<td>The device reported an error while attempting to output the text.</td>
</tr>
<tr>
<td>EFI_UNSUPPORTED</td>
<td>The output device’s mode is not currently in a defined text mode.</td>
</tr>
<tr>
<td>EFI_WARN_UNKNOWN_GLYPH</td>
<td>This warning code indicates that some of the characters in the string could not be rendered and were skipped.</td>
</tr>
</tbody>
</table>

`EFI_SIMPLE_TEXT_OUTPUT_PROTOCOL.TestString()`

Summary
Verifies that all characters in a string can be output to the target device.
Prototype

```c
typedef EFI_STATUS
(EFIAPI *EFI_TEXT_TEST_STRING) (
    IN EFI_SIMPLE_TEXT_OUTPUT_PROTOCOL *This,
    IN CHAR16 *String
);
```

Parameters

- **This**: A pointer to the EFI_SIMPLE_TEXT_OUTPUT_PROTOCOL instance. Type EFI_SIMPLE_TEXT_OUTPUT_PROTOCOL is defined in the “Related Definitions” of Section 12.4.
- **String**: The Null-terminated string to be examined for the output device(s).

Description

The TestString() function verifies that all characters in a string can be output to the target device. This function provides a way to know if the desired character codes are supported for rendering on the output device(s). This allows the installation procedure (or EFI image) to at least select character codes that the output devices are capable of displaying. Since the output device(s) may be changed between boots, if the loader cannot adapt to such changes it is recommended that the loader call OutputString() with the text it has and ignore any “unsupported” error codes. Devices that are capable of displaying the Unicode character codes will do so.

Status Codes Returned

<table>
<thead>
<tr>
<th>Code</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>EFI_SUCCESS</td>
<td>The device(s) are capable of rendering the output string.</td>
</tr>
<tr>
<td>EFI_UNSUPPORTED</td>
<td>Some of the characters in the string cannot be rendered by one or more of the output devices mapped by the EFI handle.</td>
</tr>
</tbody>
</table>

EFI_SIMPLE_TEXT_OUTPUT_PROTOCOL.QueryMode()

Summary

Returns information for an available text mode that the output device(s) supports.

Prototype

```c
typedef EFI_STATUS
(EFIAPI *EFI_TEXT_QUERY_MODE) (
    IN EFI_SIMPLE_TEXT_OUTPUT_PROTOCOL *This,
    IN UINTN ModeNumber,
    OUT UINTN *Columns,
    OUT UINTN *Rows
);
```
Parameters

This A pointer to the EFI_SIMPLE_TEXT_OUTPUT_PROTOCOL instance. Type EFI_SIMPLE_TEXT_OUTPUT_PROTOCOL is defined in the “Related Definitions” of Section 12.4.

ModeNumber The mode number to return information on.

Columns, Rows Returns the geometry of the text output device for the request ModeNumber.

Description

The QueryMode() function returns information for an available text mode that the output device(s) supports.

It is required that all output devices support at least 80x25 text mode. This mode is defined to be mode 0. If the output devices support 80x50, that is defined to be mode 1. All other text dimensions supported by the device will follow as modes 2 and above. If an output device supports modes 2 and above, but does not support 80x50, then querying for mode 1 will return EFI_UNSUPPORTED.

Status Codes Returned

<table>
<thead>
<tr>
<th>Code</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>EFI_SUCCESS</td>
<td>The requested mode information was returned.</td>
</tr>
<tr>
<td>EFI_DEVICE_ERROR</td>
<td>The device had an error and could not complete the request.</td>
</tr>
<tr>
<td>EFI_UNSUPPORTED</td>
<td>The mode number was not valid.</td>
</tr>
</tbody>
</table>

EFI_SIMPLE_TEXT_OUTPUT_PROTOCOL.SetMode()

Summary

Sets the output device(s) to a specified mode.

Prototype

```c
typedef
EFI_STATUS
(* EFIAPI EFI_TEXT_SET_MODE) (
  IN EFI_SIMPLE_TEXT_OUTPUT_PROTOCOL *This,
  IN UINTN                ModeNumber
);
```

Parameters

This A pointer to the EFI_SIMPLE_TEXT_OUTPUT_PROTOCOL instance. Type EFI_SIMPLE_TEXT_OUTPUT_PROTOCOL is defined in the “Related Definitions” of Section 12.4.

ModeNumber The text mode to set.

Description

The SetMode() function sets the output device(s) to the requested mode. On success the device is in the geometry for the requested mode, and the device has been cleared to the current background color with the cursor at (0,0).
Status Codes Returned

<table>
<thead>
<tr>
<th>Code</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>EFI_SUCCESS</td>
<td>The requested text mode was set.</td>
</tr>
<tr>
<td>EFIDEVICE_ERROR</td>
<td>The device had an error and could not complete the request.</td>
</tr>
<tr>
<td>EFI_UNSUPPORTED</td>
<td>The mode number was not valid.</td>
</tr>
</tbody>
</table>

EFI_SIMPLE_TEXT_OUTPUT_PROTOCOL.SetAttribute()

Summary
Sets the background and foreground colors for the `OutputString()` and `ClearScreen()` functions.

Prototype

typedef EFI_STATUS (EFIAPIC *EFI_TEXT_SET_ATTRIBUTE) (
    IN EFI_SIMPLE_TEXT_OUTPUT_PROTOCOL *This,
    IN UINTN Attribute
);

Parameters

This
A pointer to the `EFI_SIMPLE_TEXT_OUTPUT_PROTOCOL` instance. Type `EFI_SIMPLE_TEXT_OUTPUT_PROTOCOL` is defined in the “Related Definitions” of Section 12.4.

Attribute
The attribute to set. Bits 0..3 are the foreground color, and bits 4..6 are the background color. All other bits are reserved. See “Related Definitions” below.

Related Definitions

```
//*******************************************************
// Attributes
//*******************************************************
#define EFI_BLACK       0x00
#define EFI_BLUE        0x01
#define EFI_GREEN       0x02
#define EFI_CYAN        0x03
#define EFI_RED         0x04
#define EFI_MAGENTA     0x05
#define EFI_BROWN       0x06
#define EFI_LIGHTGRAY   0x07
#define EFI_BRIGHT      0x08
#define EFI_DARKGRAY(EFI_BLACK | EFI_BRIGHT) 0x08
#define EFI_LIGHTBLUE   0x09
#define EFI_LIGHTGREEN  0x0A
#define EFI_LIGHTCYAN   0x0B
#define EFI_LIGHTRED    0x0C
```
#define EFI_LIGHTMAGENTA 0x0D
#define EFI_YELLOW 0x0E
#define EFI_WHITE 0x0F

#define EFI_BACKGROUND_BLACK 0x00
#define EFI_BACKGROUND_BLUE 0x10
#define EFI_BACKGROUND_GREEN 0x20
#define EFI_BACKGROUND_CYAN 0x30
#define EFI_BACKGROUND_RED 0x40
#define EFI_BACKGROUND_MAGENTA 0x50
#define EFI_BACKGROUND_BROWN 0x60
#define EFI_BACKGROUND_LIGHTGRAY 0x70

// Macro to accept color values in their raw form to create
// a value that represents both a foreground and background
// color in a single byte.
// For Foreground, and EFI_* value is valid from EFI_BLACK(0x00)
// to EFI_WHITE (0x0F).
// For Background, only EFI_BLACK, EFI_BLUE, EFI_GREEN,
// EFI_CYAN, EFI_REDF, EFI_MAGENTA, EFI_BROWN, and EFI_LIGHTGRAY
// are acceptable.
//
// Do not use EFI_BACKGROUND_xxx values with this macro.
#elif define EFI_TEXT_ATTR(Foreground, Background) 
((Foreground) | ((Background) << 4))

Description
The SetAttribute() function sets the background and foreground colors for the OutputString() and ClearScreen() functions.

The color mask can be set even when the device is in an invalid text mode.

Devices supporting a different number of text colors are required to emulate the above colors to the best of the device’s capabilities.

Status Codes Returned

<table>
<thead>
<tr>
<th>Status Code</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>EFI_SUCCESS</td>
<td>The requested attributes were set.</td>
</tr>
<tr>
<td>EFI_DEVICE_ERROR</td>
<td>The device had an error and could not complete the request.</td>
</tr>
</tbody>
</table>

EFI_SIMPLE_TEXT_OUTPUT_PROTOCOL.ClearScreen()

Summary
Clears the output device(s) display to the currently selected background color.
Prototype

typedef
EFI_STATUS
(EIFIAPI *EFI_TEXT_CLEAR_SCREEN) (  
  IN EFI_SIMPLE_TEXT_OUTPUT_PROTOCOL  *This
);

Parameters

  This

A pointer to the EFI_SIMPLE_TEXT_OUTPUT_PROTOCOL instance. Type EFI_SIMPLE_TEXT_OUTPUT_PROTOCOL is defined in the “Related Definitions” of Section 12.4.

Description

The ClearScreen() function clears the output device(s) display to the currently selected background color. The cursor position is set to (0, 0).

Status Codes Returned

<table>
<thead>
<tr>
<th>Code</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>EFI_SUCCESS</td>
<td>The operation completed successfully.</td>
</tr>
<tr>
<td>EFI_DEVICE_ERROR</td>
<td>The device had an error and could not complete the request.</td>
</tr>
<tr>
<td>EFI_UNSUPPORTED</td>
<td>The output device is not in a valid text mode.</td>
</tr>
</tbody>
</table>

EFI_SIMPLE_TEXT_OUTPUT_PROTOCOL.SetCursorPosition()

Summary

Sets the current coordinates of the cursor position.

 Prototype

typedef
EFI_STATUS
(EIFIAPI *EFI_TEXT_SET_CURSOR_POSITION) (  
  IN EFI_SIMPLE_TEXT_OUTPUT_PROTOCOL  *This,
  IN UINTN   Column,
  IN UINTN   Row
);  

Parameters

  This

A pointer to the EFI_SIMPLE_TEXT_OUTPUT_PROTOCOL instance. Type EFI_SIMPLE_TEXT_OUTPUT_PROTOCOL is defined in the “Related Definitions” of Section 12.4.

  Column, Row

The position to set the cursor to. Must greater than or equal to zero and less than the number of columns and rows returned by QueryMode().
Description
The `SetCursorPosition()` function sets the current coordinates of the cursor position. The upper left corner of the screen is defined as coordinate (0, 0).

Status Codes Returned

<table>
<thead>
<tr>
<th>Status Code</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>EFI_SUCCESS</td>
<td>The operation completed successfully.</td>
</tr>
<tr>
<td>EFI_DEVICE_ERROR</td>
<td>The device had an error and could not complete the request.</td>
</tr>
<tr>
<td>EFI_UNSUPPORTED</td>
<td>The output device is not in a valid text mode, or the cursor position is invalid for the current mode.</td>
</tr>
</tbody>
</table>

**EFI_SIMPLE_TEXT_OUTPUT_PROTOCOL.EnableCursor()**

Summary
Makes the cursor visible or invisible.

Prototype
```c
typedef
    EFI_STATUS
    (EFIAPI *EFI_TEXT_ENABLE_CURSOR) (  
    IN EFI_SIMPLE_TEXT_OUTPUT_PROTOCOL *This,
    IN BOOLEAN Visible)
```

Parameters
- **This**
  A pointer to the `EFI_SIMPLE_TEXT_OUTPUT_PROTOCOL` instance. Type `EFI_SIMPLE_TEXT_OUTPUT_PROTOCOL` is defined in the “Related Definitions” of Section 12.4.
- **Visible**
  If `TRUE`, the cursor is set to be visible. If `FALSE`, the cursor is set to be invisible.

Description
The `EnableCursor()` function makes the cursor visible or invisible.

Status Codes Returned

<table>
<thead>
<tr>
<th>Status Code</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>EFI_SUCCESS</td>
<td>The operation completed successfully.</td>
</tr>
<tr>
<td>EFI_DEVICE_ERROR</td>
<td>The device had an error and could not complete the request or the device does not support changing the cursor mode.</td>
</tr>
<tr>
<td>EFI_UNSUPPORTED</td>
<td>The output device does not support visibility control of the cursor.</td>
</tr>
</tbody>
</table>

12.5 Simple Pointer Protocol
This section defines the Simple Pointer Protocol and a detailed description of the `EFI_SIMPLE_POINTER_PROTOCOL`. The intent of this section is to specify a simple method for accessing pointer devices. This would include devices such as mice and trackballs.
The **EFI_SIMPLE_POINTER_PROTOCOL** allows information about a pointer device to be retrieved. This would include the status of buttons and the motion of the pointer device since the last time it was accessed. This protocol is attached the device handle of a pointer device, and can be used for input from the user in the preboot environment.

**EFI_SIMPLE_POINTER_PROTOCOL**

**Summary**
Provides services that allow information about a pointer device to be retrieved.

**GUID**

```
#define EFI_SIMPLE_POINTER_PROTOCOL_GUID \
{0x31878c87,0xb75,0x11d5,\ 
 {0x9a,0x4f,0x00,0x90,0x27,0x3f,0xc1,0x4d}}
```

**Protocol Interface Structure**

```
typedef struct _EFI_SIMPLE_POINTER_PROTOCOL {
  EFI_SIMPLE_POINTER_RESET Reset;
  EFI_SIMPLE_POINTER_GET_STATE GetState;
  EFI_EVENT WaitForInput;
  EFI_SIMPLE_INPUT_MODE *Mode;
} EFI_SIMPLE_POINTER_PROTOCOL;
```

**Parameters**

- **Reset** Resets the pointer device. See the **Reset()** function description.
- **GetState** Retrieves the current state of the pointer device. See the **GetState()** function description.
- **WaitForInput** Event to use with **EFI_BOOT_SERVICES.WaitForEvent()** to wait for input from the pointer device.
- **Mode** Pointer to **EFI_SIMPLE_POINTER_MODE** data. The type **EFI_SIMPLE_POINTER_MODE** is defined in “Related Definitions” below.

**Related Definitions**

```c
//***************************************************************
// EFI_SIMPLE_POINTER_MODE
//***************************************************************
typedef struct {
  UINT64 ResolutionX;
  UINT64 ResolutionY;
  UINT64 ResolutionZ;
  BOOLEAN LeftButton;
  BOOLEAN RightButton;
} EFI_SIMPLE_POINTER_MODE;
```
The following data values in the `EFI_SIMPLE_POINTER_MODE` interface are read-only and are changed by using the appropriate interface functions:

- **ResolutionX**
  - The resolution of the pointer device on the x-axis in counts/mm. If 0, then the pointer device does not support an x-axis.

- **ResolutionY**
  - The resolution of the pointer device on the y-axis in counts/mm. If 0, then the pointer device does not support a y-axis.

- **ResolutionZ**
  - The resolution of the pointer device on the z-axis in counts/mm. If 0, then the pointer device does not support a z-axis.

- **LeftButton**
  - `TRUE` if a left button is present on the pointer device. Otherwise `FALSE`.

- **RightButton**
  - `TRUE` if a right button is present on the pointer device. Otherwise `FALSE`.

**Description**

The `EFI_SIMPLE_POINTER_PROTOCOL` provides a set of services for a pointer device that can used as an input device from an application written to this specification. The services include the ability to reset the pointer device, retrieve the state of the pointer device, and retrieve the capabilities of the pointer device.

**EFI_SIMPLE_POINTER_PROTOCOL.Reset()**

**Summary**

Resets the pointer device hardware.

**Prototype**

```c
typedef EFI_STATUS (EFIAPI *EFI_SIMPLE_POINTER_RESET) (
    IN EFI_SIMPLE_POINTER_PROTOCOL *This,
    IN BOOLEAN ExtendedVerification
);
```

**Parameters**

- **This**
  - A pointer to the `EFI_SIMPLE_POINTER_PROTOCOL` instance. Type `EFI_SIMPLE_POINTER_PROTOCOL` is defined in Section 12.5.

- **ExtendedVerification**
  - Indicates that the driver may perform a more exhaustive verification operation of the device during reset.

**Description**

This `Reset()` function resets the pointer device hardware.

As part of initialization process, the firmware/device will make a quick but reasonable attempt to verify that the device is functioning. If the `ExtendedVerification` flag is `TRUE` the firmware may take an extended amount of time to verify the device is operating on reset. Otherwise the reset operation is to occur as quickly as possible.
The hardware verification process is not defined by this specification and is left up to the platform firmware or driver to implement.

### Status Codes Returned

<table>
<thead>
<tr>
<th>Code</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>EFI_SUCCESS</td>
<td>The device was reset.</td>
</tr>
<tr>
<td>EFI_DEVICE_ERROR</td>
<td>The device is not functioning correctly and could not be reset.</td>
</tr>
</tbody>
</table>

### EFI_SIMPLE_POINTER_PROTOCOL.GetState()

**Summary**

Retrieves the current state of a pointer device.

**Prototype**

```c
typedef EFI_STATUS (EFIAPICALLTYPE *EFI_SIMPLE_POINTER_GET_STATE)(
    IN EFI_SIMPLE_POINTER_PROTOCOL *This,
    OUT EFI_SIMPLE_POINTER_STATE *State
);
```

**Parameters**

- `This` A pointer to the EFI_SIMPLE_POINTER_PROTOCOL instance. Type EFI_SIMPLE_POINTER_PROTOCOL is defined in Section 12.5.
- `State` A pointer to the state information on the pointer device. Type EFI_SIMPLE_POINTER_STATE is defined in “Related Definitions” below.

**Related Definitions**

```c
//*******************************************************
// EFI_SIMPLE_POINTER_STATE
//*******************************************************
typedef struct {
    INT32 RelativeMovementX;
    INT32 RelativeMovementY;
    INT32 RelativeMovementZ;
    BOOLEAN LeftButton;
    BOOLEAN RightButton;
} EFI_SIMPLE_POINTER_STATE;
```

- `RelativeMovementX` The signed distance in counts that the pointer device has been moved along the x-axis. The actual distance moved is RelativeMovementX / ResolutionX millimeters. If the ResolutionX field of the EFI_SIMPLE_POINTER_MODE structure is 0, then this pointer device does not support an x-axis, and this field must be ignored.
**RelativeMovementY**  
The signed distance in counts that the pointer device has been moved along the y-axis. The actual distance moved is

\[
\text{RelativeMovementY} / \text{ResolutionY} \text{ millimeters.}
\]

If the \text{ResolutionY} field of the \text{EFI_SIMPLE POINTER MODE} structure is 0, then this pointer device does not support a y-axis, and this field must be ignored.

**RelativeMovementZ**  
The signed distance in counts that the pointer device has been moved along the z-axis. The actual distance moved is

\[
\text{RelativeMovementZ} / \text{ResolutionZ} \text{ millimeters.}
\]

If the \text{ResolutionZ} field of the \text{EFI_SIMPLE POINTER MODE} structure is 0, then this pointer device does not support a z-axis, and this field must be ignored.

**LeftButton**  
If \text{TRUE}, then the left button of the pointer device is being pressed. If \text{FALSE}, then the left button of the pointer device is not being pressed. If the \text{LeftButton} field of the \text{EFI_SIMPLE POINTER MODE} structure is \text{FALSE}, then this field is not valid, and must be ignored.

**RightButton**  
If \text{TRUE}, then the right button of the pointer device is being pressed. If \text{FALSE}, then the right button of the pointer device is not being pressed. If the \text{RightButton} field of the \text{EFI_SIMPLE POINTER MODE} structure is \text{FALSE}, then this field is not valid, and must be ignored.

**Description**  
The \text{GetState()} function retrieves the current state of a pointer device. This includes information on the buttons associated with the pointer device and the distance that each of the axes associated with the pointer device has been moved. If the state of the pointer device has not changed since the last call to \text{GetState()}, then \text{EFI_NOT_READY} is returned. If the state of the pointer device has changed since the last call to \text{GetState()}, then the state information is placed in \text{State}, and \text{EFI_SUCCESS} is returned. If a device error occurs while attempting to retrieve the state information, then \text{EFI_DEVICE_ERROR} is returned.

**Status Codes Returned**

<table>
<thead>
<tr>
<th>Code</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>\text{EFI_SUCCESS}</td>
<td>The state of the pointer device was returned in \text{State}.</td>
</tr>
<tr>
<td>\text{EFI_NOT_READY}</td>
<td>The state of the pointer device has not changed since the last call to \text{GetState()}.</td>
</tr>
<tr>
<td>\text{EFI_DEVICE_ERROR}</td>
<td>A device error occurred while attempting to retrieve the pointer device's current state.</td>
</tr>
</tbody>
</table>

**12.6 EFI Simple Pointer Device Paths**

An \text{EFI_SIMPLE POINTER_PROTOCOL} must be installed on a handle for its services to be available to drivers and applications written to this specification. In addition to the \text{EFI_SIMPLE_POINTER_PROTOCOL}, an \text{EFI Device Path Protocol} must also be installed on the same handle. See Section 10.2 for a detailed description of the \text{EFI Device Path Protocol}.

A device path describes the location of a hardware component in a system from the processor’s point of view. This includes the list of busses that lie between the processor and the pointer controller. The \text{UEFI Specification} takes advantage of the \text{ACPI Specification} to name system components. The following set of examples shows sample device paths for a PS/2* mouse, a serial mouse, and a USB mouse.
Table 12-5 shows an example device path for a PS/2 mouse that is located behind a PCI to ISA bridge that is located at PCI device number 0x07 and PCI function 0x00, and is directly attached to a PCI root bridge. This device path consists of an ACPI Device Path Node for the PCI Root Bridge, a PCI Device Path Node for the PCI to ISA bridge, an ACPI Device Path Node for the PS/2 mouse, and a Device Path End Structure. The _HID and _UID of the first ACPI Device Path Node must match the ACPI table description of the PCI Root Bridge. The shorthand notation for this device path is:

ACPI(PNP0A03,0)/PCI(7,0)/ACPI(PNP0F03,0)

Table 12-5 PS/2 Mouse Device Path

<table>
<thead>
<tr>
<th>Byte Offset</th>
<th>Byte Length</th>
<th>Data</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x00</td>
<td>0x01</td>
<td>0x02</td>
<td>Generic Device Path Header – Type ACPI Device Path</td>
</tr>
<tr>
<td>0x01</td>
<td>0x01</td>
<td>0x01</td>
<td>Sub type – ACPI Device Path</td>
</tr>
<tr>
<td>0x02</td>
<td>0x02</td>
<td>0x0C</td>
<td>Length – 0x0C bytes</td>
</tr>
<tr>
<td>0x04</td>
<td>0x04</td>
<td>0x41D0, 0x0A03</td>
<td>_HID PNP0A03 – 0x41D0 represents a compressed string ‘PNP’ and is in the low order bytes. The compression method is described in the ACPI Specification.</td>
</tr>
<tr>
<td>0x08</td>
<td>0x04</td>
<td>0x0000</td>
<td>_UID</td>
</tr>
<tr>
<td>0x0C</td>
<td>0x01</td>
<td>0x01</td>
<td>Generic Device Path Header – Type Hardware Device Path</td>
</tr>
<tr>
<td>0x0D</td>
<td>0x01</td>
<td>0x01</td>
<td>Sub type – PCI</td>
</tr>
<tr>
<td>0x0E</td>
<td>0x02</td>
<td>0x06</td>
<td>Length – 0x06 bytes</td>
</tr>
<tr>
<td>0x10</td>
<td>0x01</td>
<td>0x00</td>
<td>PCI Function</td>
</tr>
<tr>
<td>0x11</td>
<td>0x01</td>
<td>0x07</td>
<td>PCI Device</td>
</tr>
<tr>
<td>0x12</td>
<td>0x01</td>
<td>0x02</td>
<td>Generic Device Path Header – Type ACPI Device Path</td>
</tr>
<tr>
<td>0x13</td>
<td>0x01</td>
<td>0x01</td>
<td>Sub type – ACPI Device Path</td>
</tr>
<tr>
<td>0x14</td>
<td>0x02</td>
<td>0x0C</td>
<td>Length – 0x0C bytes</td>
</tr>
<tr>
<td>0x16</td>
<td>0x04</td>
<td>0x41D0, 0x0F03</td>
<td>_HID PNP0A03 – 0x41D0 represents a compressed string ‘PNP’ and is in the low order bytes. The compression method is described in the ACPI Specification.</td>
</tr>
<tr>
<td>0x1A</td>
<td>0x04</td>
<td>0x0000</td>
<td>_UID</td>
</tr>
<tr>
<td>0x1E</td>
<td>0x01</td>
<td>0xFF</td>
<td>Generic Device Path Header – Type End of Hardware Device Path</td>
</tr>
<tr>
<td>0x1F</td>
<td>0x01</td>
<td>0xFF</td>
<td>Sub type – End of Entire Device Path</td>
</tr>
<tr>
<td>0x20</td>
<td>0x02</td>
<td>0x04</td>
<td>Length – 0x04 bytes</td>
</tr>
</tbody>
</table>

Table 12-6 shows an example device path for a serial mouse that is located on COM 1 behind a PCI to ISA bridge that is located at PCI device number 0x07 and PCI function 0x00. The PCI to ISA bridge is directly attached to a PCI root bridge, and the communications parameters for COM 1 are 1200 baud, no parity, 8 data bits, and 1 stop bit. This device path consists of an ACPI Device Path Node for the PCI Root Bridge, a PCI Device Path Node for the PCI to ISA bridge, an ACPI Device Path Node for COM 1, a UART Device Path Node for the communications parameters, an ACPI Device Path Node for the serial mouse, and a Device Path End Structure. The _HID and _UID of the first ACPI Device Path Node must match the ACPI table description of the PCI Root Bridge. The shorthand notation for this device path is:
### ACPI(PNP0A03,0)/PCI(7,0)/ACPI(PNP0501,0)/UART(1200,N,8,1)/ACPI(PNP0F01,0)

#### Table 12-6 Serial Mouse Device Path

<table>
<thead>
<tr>
<th>Byte Offset</th>
<th>Byte Length</th>
<th>Data</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x00</td>
<td>0x01</td>
<td>0x02</td>
<td><strong>Generic Device Path Header</strong> – Type ACPI Device Path</td>
</tr>
<tr>
<td>0x01</td>
<td>0x01</td>
<td>0x01</td>
<td>Sub type – ACPI Device Path</td>
</tr>
<tr>
<td>0x02</td>
<td>0x02</td>
<td>0x0C</td>
<td>Length – 0x0C bytes</td>
</tr>
<tr>
<td>0x04</td>
<td>0x04</td>
<td>0x41D0, 0x0A03</td>
<td>_HID PNP0A03 – 0x41D0 represents the compressed string ‘PNP’ and is encoded in the low order bytes. The compression method is described in the ACPI Specification.</td>
</tr>
<tr>
<td>0x08</td>
<td>0x04</td>
<td>0x0000</td>
<td>_UID</td>
</tr>
<tr>
<td>0x0C</td>
<td>0x01</td>
<td>0x01</td>
<td><strong>Generic Device Path Header</strong> – Type Hardware Device Path</td>
</tr>
<tr>
<td>0x0D</td>
<td>0x01</td>
<td>0x01</td>
<td>Sub type – PCI</td>
</tr>
<tr>
<td>0x0E</td>
<td>0x02</td>
<td>0x06</td>
<td>Length – 0x06 bytes</td>
</tr>
<tr>
<td>0x10</td>
<td>0x01</td>
<td>0x00</td>
<td>PCI Function</td>
</tr>
<tr>
<td>0x11</td>
<td>0x01</td>
<td>0x07</td>
<td>PCI Device</td>
</tr>
<tr>
<td>0x12</td>
<td>0x01</td>
<td>0x02</td>
<td><strong>Generic Device Path Header</strong> – Type ACPI Device Path</td>
</tr>
<tr>
<td>0x13</td>
<td>0x01</td>
<td>0x01</td>
<td>Sub type – ACPI Device Path</td>
</tr>
<tr>
<td>0x14</td>
<td>0x02</td>
<td>0x0C</td>
<td>Length – 0x0C bytes</td>
</tr>
<tr>
<td>0x16</td>
<td>0x04</td>
<td>0x41D0, 0x0501</td>
<td>_HID PNP0501 – 0x41D0 represents the compressed string ‘PNP’ and is encoded in the low order bytes. The compression method is described in the ACPI Specification.</td>
</tr>
<tr>
<td>0x1A</td>
<td>0x04</td>
<td>0x0000</td>
<td>_UID</td>
</tr>
<tr>
<td>0x1E</td>
<td>0x01</td>
<td>0x03</td>
<td><strong>Generic Device Path Header</strong> – Messaging Device Path</td>
</tr>
<tr>
<td>0x1F</td>
<td>0x01</td>
<td>0x0E</td>
<td>Sub type – UART Device Path</td>
</tr>
<tr>
<td>0x20</td>
<td>0x02</td>
<td>0x13</td>
<td>Length – 0x13 bytes</td>
</tr>
<tr>
<td>0x22</td>
<td>0x04</td>
<td>0x00</td>
<td>Reserved</td>
</tr>
<tr>
<td>0x26</td>
<td>0x08</td>
<td>1200</td>
<td>Baud Rate</td>
</tr>
<tr>
<td>0x2E</td>
<td>0x01</td>
<td>0x08</td>
<td>Data Bits</td>
</tr>
<tr>
<td>0x2F</td>
<td>0x01</td>
<td>0x01</td>
<td>Parity</td>
</tr>
<tr>
<td>0x30</td>
<td>0x01</td>
<td>0x01</td>
<td>Stop Bits</td>
</tr>
<tr>
<td>0x31</td>
<td>0x01</td>
<td>0x02</td>
<td><strong>Generic Device Path Header</strong> – Type ACPI Device Path</td>
</tr>
<tr>
<td>0x32</td>
<td>0x01</td>
<td>0x01</td>
<td>Sub type – ACPI Device Path</td>
</tr>
<tr>
<td>0x33</td>
<td>0x02</td>
<td>0x0C</td>
<td>Length – 0x0C bytes</td>
</tr>
<tr>
<td>0x35</td>
<td>0x04</td>
<td>0x41D0, 0x0F01</td>
<td>_HID PNP0F01 – 0x41D0 represents the compressed string ‘PNP’ and is encoded in the low order bytes. The compression method is described in the ACPI Specification.</td>
</tr>
<tr>
<td>0x39</td>
<td>0x04</td>
<td>0x0000</td>
<td>_UID</td>
</tr>
<tr>
<td>0x3D</td>
<td>0x01</td>
<td>0xFF</td>
<td><strong>Generic Device Path Header</strong> – Type End of Hardware Device Path</td>
</tr>
<tr>
<td>0x3E</td>
<td>0x01</td>
<td>0xFF</td>
<td>Sub type – End of Entire Device Path</td>
</tr>
<tr>
<td>0x3F</td>
<td>0x02</td>
<td>0x04</td>
<td>Length – 0x04 bytes</td>
</tr>
</tbody>
</table>
Table 12-7 shows an example device path for a USB mouse that is behind a PCI to USB host controller that is located at PCI device number 0x07 and PCI function 0x02. The PCI to USB host controller is directly attached to a PCI root bridge. This device path consists of an ACPI Device Path Node for the PCI Root Bridge, a PCI Device Path Node for the PCI to USB controller, a USB Device Path Node, and a Device Path End Structure. The _HID and _UID of the first ACPI Device Path Node must match the ACPI table description of the PCI Root Bridge. The shorthand notation for this device path is:

**ACPI(PNP0A03,0)/PCI(7,2)/USB(0,0)**

Table 12-7 USB Mouse Device Path

<table>
<thead>
<tr>
<th>Byte Offset</th>
<th>Byte Length</th>
<th>Data</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x00</td>
<td>0x01</td>
<td>0x02</td>
<td>Generic Device Path Header – Type ACPI Device Path</td>
</tr>
<tr>
<td>0x01</td>
<td>0x01</td>
<td>0x01</td>
<td>Sub type – ACPI Device Path</td>
</tr>
<tr>
<td>0x02</td>
<td>0x02</td>
<td>0x0C</td>
<td>Length – 0x0C bytes</td>
</tr>
<tr>
<td>0x04</td>
<td>0x04</td>
<td>0x41D0, 0x0A03</td>
<td>_HID PNP0A03 – 0x41D0 represents a compressed string ‘PNP’ and is in the low order bytes.</td>
</tr>
<tr>
<td>0x08</td>
<td>0x04</td>
<td>0x0000</td>
<td>_UID</td>
</tr>
<tr>
<td>0x0C</td>
<td>0x01</td>
<td>0x01</td>
<td>Generic Device Path Header – Type Hardware Device Path</td>
</tr>
<tr>
<td>0x0D</td>
<td>0x01</td>
<td>0x01</td>
<td>Sub type – PCI</td>
</tr>
<tr>
<td>0x0E</td>
<td>0x02</td>
<td>0x06</td>
<td>Length – 0x06 bytes</td>
</tr>
<tr>
<td>0x10</td>
<td>0x01</td>
<td>0x02</td>
<td>PCI Function</td>
</tr>
<tr>
<td>0x11</td>
<td>0x01</td>
<td>0x07</td>
<td>PCI Device</td>
</tr>
<tr>
<td>0x12</td>
<td>0x01</td>
<td>0x03</td>
<td>Generic Device Path Header – Type Messaging Device Path</td>
</tr>
<tr>
<td>0x13</td>
<td>0x01</td>
<td>0x05</td>
<td>Sub type – USB</td>
</tr>
<tr>
<td>0x14</td>
<td>0x02</td>
<td>0x06</td>
<td>Length – 0x06 bytes</td>
</tr>
<tr>
<td>0x16</td>
<td>0x01</td>
<td>0x00</td>
<td>USB Port Number</td>
</tr>
<tr>
<td>0x17</td>
<td>0x01</td>
<td>0x00</td>
<td>USB Endpoint Number</td>
</tr>
<tr>
<td>0x18</td>
<td>0x01</td>
<td>0xFF</td>
<td>Generic Device Path Header – Type End of Hardware Device Path</td>
</tr>
<tr>
<td>0x19</td>
<td>0x01</td>
<td>0xFF</td>
<td>Sub type – End of Entire Device Path</td>
</tr>
<tr>
<td>0x1A</td>
<td>0x02</td>
<td>0x04</td>
<td>Length – 0x04 bytes</td>
</tr>
</tbody>
</table>

### 12.7 Absolute Pointer Protocol

This section defines the Absolute Pointer Protocol and a detailed description of the `EFI_ABSOLUTE_POINTER_PROTOCOL`. The intent of this section is to specify a simple method for accessing absolute pointer devices. This would include devices like touch screens, and digitizers.

The `EFI_ABSOLUTE_POINTER_PROTOCOL` allows information about a pointer device to be retrieved. This would include the status of buttons and the coordinates of the pointer device on the last time it was activated. This protocol is attached to the device handle of an absolute pointer device, and can be used for input from the user in the preboot environment.
Supported devices may return 1, 2, or 3 axis of information. The Z axis may optionally be used to return pressure data measurements derived from user pen force.

All supported devices must support a touch-active status. Supported devices may optionally support a second input button, for example a pen side-button.

**EFI_ABSOLUTE_POINTER_PROTOCOL**

*Summary*

Provides services that allow information about an absolute pointer device to be retrieved.

**GUID**

```c
#define EFI_ABSOLUTE_POINTER_PROTOCOL_GUID \
{0x8D59D32B, 0xC655, 0x4AE9, \
 {0x9B, 0x15, 0xF2, 0x59, 0x04, 0x99, 0x2A, 0x43}}
```

**Protocol Interface Structure**

```c
typedef struct _EFI_ABSOLUTE_POINTER_PROTOCOL {
  EFI_ABSOLUTE_POINTER_RESET     Reset;
  EFI_ABSOLUTE_POINTER_GET_STATE GetState;
  EFI_EVENT                      WaitForInput;
  EFI_ABSOLUTE_POINTER_MODE     *Mode;
} EFI_ABSOLUTE_POINTER_PROTOCOL;
```

**Parameters**

- **Reset**
  Resets the pointer device. See the `Reset()` function description.

- **GetState**
  Retrieves the current state of the pointer device. See the `GetState()` function description.

- **WaitForInput**
  Event to use with `WaitForEvent()` to wait for input from the pointer device.

- ***Mode**
  Pointer to `EFI_ABSOLUTE_POINTER_MODE` data. The type `EFI_ABSOLUTE_POINTER_MODE` is defined in "Related Definitions" below.
Related Definitions

```c
typedef struct {
    UINT64 AbsoluteMinX;
    UINT64 AbsoluteMinY;
    UINT64 AbsoluteMinZ;
    UINT64 AbsoluteMaxX;
    UINT64 AbsoluteMaxY;
    UINT64 AbsoluteMaxZ;
    UINT32 Attributes;
} EFI_ABSOLUTE_POINTER_MODE;
```

The following data values in the `EFI_ABSOLUTE_POINTER_MODE` interface are read-only and are changed by using the appropriate interface functions:

- **AbsoluteMinX**: The Absolute Minimum of the device on the x-axis.
- **AbsoluteMinY**: The Absolute Minimum of the device on the y-axis.
- **AbsoluteMinZ**: The Absolute Minimum of the device on the z-axis.
- **AbsoluteMaxX**: The Absolute Maximum of the device on the x-axis. If 0, and the `AbsoluteMinX` is 0, then the pointer device does not support a x-axis.
- **AbsoluteMaxY**: The Absolute Maximum of the device on the y-axis. If 0, and the `AbsoluteMinY` is 0, then the pointer device does not support a y-axis.
- **AbsoluteMaxZ**: The Absolute Maximum of the device on the z-axis. If 0, and the `AbsoluteMinZ` is 0, then the pointer device does not support a z-axis.
- **Attributes**: The following bits are set as needed (or'd together) to indicate the capabilities of the device supported. The remaining bits are undefined and should be returned as 0.

```
#define EFI_ABSP_SupportsAltActive    0x00000001
#define EFI_ABSP_SupportsPressureAsZ  0x00000002
```

**EFI_ABSP_SupportsAltActive**
- If set, indicates this device supports an alternate button input.

**EFI_ABSP_SupportsPressureAsZ**
- If set, indicates this device returns pressure data in parameter `CurrentZ`.

The driver is not permitted to return all zeros for all three pairs of Min and Max as this would indicate no axis supported.

**Description**

The `EFI_ABSOLUTE_POINTER_PROTOCOL` provides a set of services for a pointer device that can be used as an input device from an application written to this specification. The services include the ability to
reset the pointer device, retrieve the state of the pointer device, and retrieve the capabilities of the pointer device. In addition certain data items describing the device are provided.

**EFI_ABSOLUTE_POINTER_PROTOCOL.Reset()**

**Summary**
Resets the pointer device hardware.

**Prototype**
```
typedef EFI_STATUS
    (EFIAPI *EFI_ABSOLUTE_POINTER_RESET) (
    IN EFI_ABSOLUTE_POINTER_PROTOCOL *This,
    IN BOOLEAN ExtendedVerification
    );
```

**Parameters**
- **This**
  A pointer to the **EFI_ABSOLUTE_POINTER_PROTOCOL** instance. Type **EFI_ABSOLUTE_POINTER_PROTOCOL** is defined in this section.
- **ExtendedVerification**
  Indicates that the driver may perform a more exhaustive verification operation of the device during reset.

**Description**
This Reset() function resets the pointer device hardware. As part of initialization process, the firmware/device will make a quick but reasonable attempt to verify that the device is functioning. If the ExtendedVerification flag is TRUE the firmware may take an extended amount of time to verify the device is operating on reset. Otherwise the reset operation is to occur as quickly as possible.

The hardware verification process is not defined by this specification and is left up to the platform firmware or driver to implement.

**Codes Returned**

<table>
<thead>
<tr>
<th>Code</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>EFI_SUCCESS</td>
<td>The device was reset.</td>
</tr>
<tr>
<td>EFI_DEVICE_ERROR</td>
<td>The device is not functioning correctly and could not be reset.</td>
</tr>
</tbody>
</table>

**EFI_ABSOLUTE_POINTER_PROTOCOL.GetState()**

**Summary**
Retrieves the current state of a pointer device.
Prototype

```c
typedef
EFI_STATUS
(EFIAPI *EFI_ABSOLUTE_POINTER_GET_STATE) (IN EFI_ABSOLUTE_POINTER_PROTOCOL *This,
OUT EFI_ABSOLUTE_POINTER_STATE *State);
```

Parameters

- **This**: A pointer to the `EFI_ABSOLUTE_POINTER_PROTOCOL` instance. Type `EFI_ABSOLUTE_POINTER_PROTOCOL` is defined in Section 12.7.
- **State**: A pointer to the state information on the pointer device. Type `EFI_ABSOLUTE_POINTER_STATE` is defined in "Related Definitions" below.

Related Definitions

```c
//******************************************************************************
// EFI_ABSOLUTE_POINTER_STATE
//******************************************************************************
typedef struct {
    UINT64 CurrentX;
    UINT64 CurrentY;
    UINT64 CurrentZ;
    UINT32 ActiveButtons;
} EFI_ABSOLUTE_POINTER_STATE;
```

- **CurrentX**: The unsigned position of the activation on the x axis. If the AbsoluteMinX and the AbsoluteMaxX fields of the `EFI_ABSOLUTE_POINTER_MODE` structure are both 0, then this pointer device does not support an x-axis, and this field must be ignored.

- **CurrentY**: The unsigned position of the activation on the y axis. If the AbsoluteMinY and the AbsoluteMaxY fields of the `EFI_ABSOLUTE_POINTER_MODE` structure are both 0, then this pointer device does not support a y-axis, and this field must be ignored.

- **CurrentZ**: The unsigned position of the activation on the z axis, or the pressure measurement. If the AbsoluteMinZ and the AbsoluteMaxZ fields of the `EFI_ABSOLUTE_POINTER_MODE` structure are both 0, then this pointer device does not support a z-axis, and this field must be ignored.

- **ActiveButtons**: Bits are set to 1 in this structure item to indicate that device buttons are active.
Related Definitions

```c
// definitions of bits within ActiveButtons
#define EFI_ABSP_TouchActive  0x00000001
#define EFI_ABS_AltActive     0x00000002
```

EFI_ABSP_TouchActive This bit is set if the touch sensor is active.
EFI_ABS_AltActive This bit is set if the alt sensor, such as pen-side button, is active.

Description

The GetState() function retrieves the current state of a pointer device. This includes information on the active state associated with the pointer device and the current position of the axes associated with the pointer device. If the state of the pointer device has not changed since the last call to `GetState()`, then EFI_NOT_READY is returned. If the state of the pointer device has changed since the last call to `GetState()`, then the state information is placed in State, and EFI_SUCCESS is returned. If a device error occurs while attempting to retrieve the state information, then EFI_DEVICE_ERROR is returned.

Status Codes Returned

<table>
<thead>
<tr>
<th>Code</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>EFI_SUCCESS</td>
<td>The state of the pointer device was returned in State.</td>
</tr>
<tr>
<td>EFI_NOT_READY</td>
<td>The state of the pointer device has not changed since the last call to <code>GetState()</code>.</td>
</tr>
<tr>
<td>EFI_DEVICE_ERROR</td>
<td>A device error occurred while attempting to retrieve the pointer device's current state.</td>
</tr>
</tbody>
</table>

12.8 Serial I/O Protocol

This section defines the Serial I/O protocol. This protocol is used to abstract byte stream devices.

EFI_SERIAL_IO_PROTOCOL

Summary

This protocol is used to communicate with any type of character-based I/O device.

GUID

```c
#define EFI_SERIAL_IO_PROTOCOL_GUID \
  {0xBB25CF6F,0xF1D4,0x11D2,\ 
   0x9a,0x0c,0x00,0x90,0x27,0x3f,0xc1,0xfd}
```

Revision Number

```c
#define EFI_SERIAL_IO_PROTOCOL_REVISION     0x00010000
#define EFI_SERIAL_IO_PROTOCOL_REVISION1p1  0x00010001
```
Protocol Interface Structure

typedef struct {
    UINT32 Revision;
    EFI_SERIAL_RESET Reset;
    EFI_SERIAL_SET_ATTRIBUTES SetAttributes;
    EFI_SERIAL_SET_CONTROL_BITS SetControl;
    EFI_SERIAL_GET_CONTROL_BITS GetControl;
    EFI_SERIAL_WRITE Write;
    EFI_SERIAL_READ Read;
    SERIAL_IO_MODE *Mode;
    CONST EFI_GUID *DeviceTypeGuid; // Revision 1.1
} EFI_SERIAL_IO_PROTOCOL;

Parameters

Revision The revision to which the EFI_SERIAL_IO_PROTOCOL adheres. All future revisions must be backwards compatible. If a future version is not backwards compatible, it is not the same GUID.

Reset Resets the hardware device.

SetAttributes Sets communication parameters for a serial device. These include the baud rate, receive FIFO depth, transmit/receive time out, parity, data bits, and stop bit attributes.

SetControl Sets the control bits on a serial device. These include Request to Send and Data Terminal Ready.

GetControl Reads the status of the control bits on a serial device. These include Clear to Send, Data Set Ready, Ring Indicator, and Carrier Detect.

Write Sends a buffer of characters to a serial device.

Read Receives a buffer of characters from a serial device.

Mode Pointer to SERIAL_IO_MODE data. Type SERIAL_IO_MODE is defined in “Related Definitions” below.

DeviceTypeGuid Pointer to a GUID identifying the device connected to the serial port. This field is NULL when the protocol is installed by the serial port driver and may be populated by a platform driver for a serial port with a known device attached. The field will remain NULL if there is no platform serial device identification information available.
Related Definitions

```c
typedef struct {
    UINT32 ControlMask;

    // current Attributes
    UINT32 Timeout;
    UINT64 BaudRate;
    UINT32 ReceiveFifoDepth;
    UINT32 DataBits;
    UINT32 Parity;
    UINT32 StopBits;
} SERIAL_IO_MODE;
```

The data values in the `SERIAL_IO_MODE` are read-only and are updated by the code that produces the `EFI_SERIAL_IO_PROTOCOL` functions:

- **ControlMask**: A mask of the Control bits that the device supports. The device must always support the Input Buffer Empty control bit.
- **Timeout**: If applicable, the number of microseconds to wait before timing out a Read or Write operation.
- **BaudRate**: If applicable, the current baud rate setting of the device; otherwise, baud rate has the value of zero to indicate that device runs at the device’s designed speed.
- **ReceiveFifoDepth**: The number of characters the device will buffer on input.
- **DataBits**: The number of data bits in each character.
- **Parity**: If applicable, this is the `EFI_PARITY_TYPE` that is computed or checked as each character is transmitted or received. If the device does not support parity the value is the default parity value.
- **StopBits**: If applicable, the `EFI_STOP_BITS_TYPE` number of stop bits per character. If the device does not support stop bits the value is the default stop bit value.
The Serial I/O protocol is used to communicate with UART-style serial devices. These can be standard UART serial ports in PC-AT systems, serial ports attached to a USB interface, or potentially any character-based I/O device.

The Serial I/O protocol can control byte I/O style devices from a generic device, to a device with features such as a UART. As such many of the serial I/O features are optional to allow for the case of devices that do not have UART controls. Each of these options is called out in the specific serial I/O functions.

The default attributes for all UART-style serial device interfaces are: 115,200 baud, a 1 byte receive FIFO, a 1,000,000 microsecond timeout per character, no parity, 8 data bits, and 1 stop bit. Flow control is the responsibility of the software that uses the protocol. Hardware flow control can be implemented through the use of the GetControl() and SetControl() functions (described below) to monitor and assert the flow control signals. The XON/XOFF flow control algorithm can be implemented in software by inserting XON and XOFF characters into the serial data stream as required.

Special care must be taken if a significant amount of data is going to be read from a serial device. Since UEFI drivers are polled mode drivers, characters received on a serial device might be missed. It is the responsibility of the software that uses the protocol to check for new data often enough to guarantee that no characters will be missed. The required polling frequency depends on the baud rate of the connection and the depth of the receive FIFO.
12.8.1 Serial Device Identification

Serial device identification is accomplished through the interaction of three distinct drivers. The serial port driver binds to the serial port hardware and produces the EFI_SERIAL_IO_PROTOCOL. At the time the protocol is produced the DeviceTypeGuid field is NULL.

During the UEFI Driver Binding process a platform driver, with a EFI_DRIVER_BINDING_PROTOCOL Version field in the range of \texttt{0xffffffff0} to \texttt{0xffffffff} can check for the presence of the EFI_SERIAL_IO_PROTOCOL and any other necessary information in Supported() to check if the serial port instance is recognized for the purposes of provide serial device identification information. If the port instance is recognized then EFI_SUCCESS will be returned from Supported(). Since the driver binding Version field is higher than any device driver the platform’s serial device identification driver binding instance will have Start() called. This function will write the DeviceTypeGuid with a value that identifies the attached serial device.

When the driver binding process continues the serial device driver can use the DeviceTypeGuid field to determine the serial device connected to the port is supported.

Serial device drivers for non-terminal devices that will co-exist with backwards-compatible terminal drivers must check that the EFI_SERIAL_IO_PROTOCOL Revision field is at least \texttt{0x00010001} and compare the DeviceTypeGuid in their driver binding Supported() function. Terminal drivers provide backwards compatibility by assuming a Terminal device is present when a protocol instance Revision is the original \texttt{0x00010000} value. Terminal drivers may also assume a Terminal device is present if the DeviceTypeGuid is NULL for cases where the platform does not provide serial identification information.

12.8.2 Serial Device Type GUIDs

```c
#define EFI_SERIAL_TERMINAL_DEVICE_TYPE_GUID  
  { 0x6ad9a60f, 0x5815, 0x4c7c, 
    { 0x8a, 0x10, 0x50, 0x53, 0xdb, 0xbf, 0x1b } }
```

The EFI_SERIAL_TERMINAL_DEVICE_TYPE_GUID describes a serial terminal type device suitable for use as a UEFI console.

Vendors may define and use additional GUIDs for other serial device types.
Figure 12-1 Serial Device Identification Driver Relationships

**EFI_SERIAL_IO_PROTOCOL.Reset()**

**Summary**
Resets the serial device.

**Prototype**
```
typedef
    EFI_STATUS
    (EFIAPI *EFI_SERIAL_RESET) (  
        IN EFI_SERIAL_IO_PROTOCOL *This
    );
```

**Parameters**
- `This` A pointer to the `EFI_SERIAL_IO_PROTOCOL` instance. Type `EFI_SERIAL_IO_PROTOCOL` is defined in [Section 12.8](#).

**Description**
The `Reset()` function resets the hardware of a serial device.
### Status Codes Returned

<table>
<thead>
<tr>
<th>Code</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>EFI_SUCCESS</td>
<td>The serial device was reset.</td>
</tr>
<tr>
<td>EFI_DEVICE_ERROR</td>
<td>The serial device could not be reset.</td>
</tr>
</tbody>
</table>

### EFI_SERIAL_IO_PROTOCOL.SetAttributes()

#### Summary
Sets the baud rate, receive FIFO depth, transmit/receive time out, parity, data bits, and stop bits on a serial device.

```c
EFI_STATUS
(EFIAPI *EFI_SERIAL_SET_ATTRIBUTES) (  
  IN EFI_SERIAL_IO_PROTOCOL *This,
  IN UINT64 BaudRate,  
  IN UINT32 ReceiveFifoDepth,  
  IN UINT32 Timeout
  IN EFI_PARITY_TYPE Parity,
  IN UINT8 DataBits,
  IN EFI_STOP_BITS_TYPE StopBits
  );
```

#### Parameters

- **This**
  A pointer to the EFI_SERIAL_IO_PROTOCOL instance. Type EFI_SERIAL_IO_PROTOCOL is defined in Section 12.8.

- **BaudRate**
  The requested baud rate. A BaudRate value of 0 will use the device’s default interface speed.

- **ReceiveFifoDepth**
  The requested depth of the FIFO on the receive side of the serial interface. A ReceiveFifoDepth value of 0 will use the device’s default FIFO depth.

- **Timeout**
  The requested time out for a single character in microseconds. This timeout applies to both the transmit and receive side of the interface. A Timeout value of 0 will use the device’s default time out value.

- **Parity**
  The type of parity to use on this serial device. A Parity value of DefaultParity will use the device’s default parity value. Type EFI_PARITY_TYPE is defined in “Related Definitions” in Section 12.8.

- **DataBits**
  The number of data bits to use on this serial device. A DataBits value of 0 will use the device’s default data bit setting.

- **StopBits**
  The number of stop bits to use on this serial device. A StopBits value of DefaultStopBits will use the device’s default number of stop bits. Type EFI_STOP_BITS_TYPE is defined in “Related Definitions” in Section 12.8.
Description

The `SetAttributes()` function sets the baud rate, receive-FIFO depth, transmit/receive time out, parity, data bits, and stop bits on a serial device.

The controller for a serial device is programmed with the specified attributes. If the `Parity`, `DataBits`, or `StopBits` values are not valid, then an error will be returned. If the specified `BaudRate` is below the minimum baud rate supported by the serial device, an error will be returned. The nearest baud rate supported by the serial device will be selected without exceeding the `BaudRate` parameter. If the specified `ReceiveFifoDepth` is below the smallest FIFO size supported by the serial device, an error will be returned. The nearest FIFO size supported by the serial device will be selected without exceeding the `ReceiveFifoDepth` parameter.

Status Codes Returned

<table>
<thead>
<tr>
<th>Code</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>EFI_SUCCESS</td>
<td>The new attributes were set on the serial device.</td>
</tr>
<tr>
<td>EFI_INVALID_PARAMETER</td>
<td>One or more of the attributes has an unsupported value.</td>
</tr>
<tr>
<td>EFI_DEVICE_ERROR</td>
<td>The serial device is not functioning correctly.</td>
</tr>
</tbody>
</table>

**EFI_SERIAL_IO_PROTOCOL.SetControl()**

Summary

Sets the control bits on a serial device.

Prototype

```c
typedef EFI_STATUS (EFIAPI *EFI_SERIAL_SET_CONTROL_BITS) (  
    IN EFI_SERIAL_IO_PROTOCOL *This,  
    IN UINT32 Control
);
```

Parameters

- **This**
  
  A pointer to the `EFI_SERIAL_IO_PROTOCOL` instance. Type `EFI_SERIAL_IO_PROTOCOL` is defined in Section 12.8.

- **Control**
  
  Sets the bits of `Control` that are settable. See “Related Definitions” below.
Related Definitions

```
//*******************************************************
// CONTROL BITS
//*******************************************************
#define EFI_SERIAL_CLEAR_TO_SEND                0x0010
#define EFI_SERIAL_DATA_SET_READY               0x0020
#define EFI_SERIAL_RING_INDICATE                0x0040
#define EFI_SERIAL_CARRIER_DETECT               0x0080
#define EFI_SERIAL_REQUEST_TO_SEND              0x0002
#define EFI_SERIAL_DATA_TERMINAL_READY          0x0001
#define EFI_SERIAL_INPUT_BUFFER_EMPTY           0x0100
#define EFI_SERIAL_OUTPUT_BUFFER_EMPTY          0x0200
#define EFI_SERIAL_HARDWARE_LOOPBACK_ENABLE     0x1000
#define EFI_SERIAL_SOFTWARE_LOOPBACK_ENABLE     0x2000
#define EFI_SERIAL_HARDWARE_FLOW_CONTROL_ENABLE 0x4000
```

Description

The **SetControl()** function is used to assert or deassert the control signals on a serial device. The following signals are set according their bit settings:

- Request to Send
- Data Terminal Ready

Only the **REQUEST_TOSEND**, **DATA_TERMINAL_READY**, **HARDWARE_LOOPBACK_ENABLE**, **SOFTWARE_LOOPBACK_ENABLE**, and **HARDWARE_FLOW_CONTROL_ENABLE** bits can be set with **SetControl()**. All the bits can be read with **GetControl()**.

Status Codes Returned

<table>
<thead>
<tr>
<th>EFI_CODE</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>EFI_SUCCESS</td>
<td>The new control bits were set on the serial device.</td>
</tr>
<tr>
<td>EFI_UNSUPPORTED</td>
<td>The serial device does not support this operation.</td>
</tr>
<tr>
<td>EFI_DEVICE_ERROR</td>
<td>The serial device is not functioning correctly.</td>
</tr>
</tbody>
</table>

**EFI_SERIAL_IO_PROTOCOL.GetControl()**

Summary

Retrieves the status of the control bits on a serial device.
Prototype

```c
typedef EFI_STATUS
  (EFI_APPI EFI_SERIAL_GET_CONTROL_BITS) (IN EFI_SERIAL_IO_PROTOCOL *This,
   OUT UINT32 *Control);
```

Parameters

- **This**
  A pointer to the `EFI_SERIAL_IO_PROTOCOL` instance. Type `EFI_SERIAL_IO_PROTOCOL` is defined in Section 12.8.

- **Control**
  A pointer to return the current control signals from the serial device. See “Related Definitions” below.

Related Definitions

```c
//******************************************************
// CONTROL BITS
//******************************************************
#define EFI_SERIAL_CLEAR_TO_SEND 0x0010
#define EFI_SERIAL_DATA_SET_READY 0x0020
#define EFI_SERIAL_RING_INDICATE 0x0040
#define EFI_SERIAL_CARRIER_DETECT 0x0080
#define EFI_SERIAL_REQUEST_TO_SEND 0x0002
#define EFI_SERIAL_DATA_TERMINAL_READY 0x0001
#define EFI_SERIAL_INPUT_BUFFER_EMPTY 0x0100
#define EFI_SERIAL_OUTPUT_BUFFER_EMPTY 0x0200
#define EFI_SERIAL_HARDWARE_LOOPBACK_ENABLE 0x1000
#define EFI_SERIAL_SOFTWARE_LOOPBACK_ENABLE 0x2000
#define EFI_SERIAL_HARDWARE_FLOW_CONTROL_ENABLE 0x4000
```

Description

The `GetControl()` function retrieves the status of the control bits on a serial device.

Status Codes Returned

<table>
<thead>
<tr>
<th>Code</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>EFI_SUCCESS</td>
<td>The control bits were read from the serial device.</td>
</tr>
<tr>
<td>EFI_DEVICE_ERROR</td>
<td>The serial device is not functioning correctly.</td>
</tr>
</tbody>
</table>

**EFI_SERIAL_IO_PROTOCOL.Write()**

Summary

Writes data to a serial device.
Prototype

typedef

EFI_STATUS

( EFIAPI *EFI_SERIAL_WRITE ) ( 

IN EFI_SERIAL_IO_PROTOCOL *This,

IN OUT UINTN *BufferSize,

IN VOID *Buffer

);

Parameters

This A pointer to the EFI_SERIAL_IO_PROTOCOL instance. Type EFI_SERIAL_IO_PROTOCOL is defined in Section 12.8.

BufferSize On input, the size of the Buffer. On output, the amount of data actually written.

Buffer The buffer of data to write.

Description

The Write() function writes the specified number of bytes to a serial device. If a time out error occurs while data is being sent to the serial port, transmission of this buffer will terminate, and EFI_TIMEOUT will be returned. In all cases the number of bytes actually written to the serial device is returned in BufferSize.

Status Codes Returned

<table>
<thead>
<tr>
<th>Status Code</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>EFI_SUCCESS</td>
<td>The data was written.</td>
</tr>
<tr>
<td>EFI_DEVICE_ERROR</td>
<td>The device reported an error.</td>
</tr>
<tr>
<td>EFI_TIMEOUT</td>
<td>The data write was stopped due to a timeout.</td>
</tr>
</tbody>
</table>

EFI_SERIAL_IO_PROTOCOL.Read()

Summary

Reads data from a serial device.

Prototype

typedef

EFI_STATUS

( EFIAPI *EFI_SERIAL_READ ) ( 

IN EFI_SERIAL_IO_PROTOCOL *This,

IN OUT UINTN *BufferSize,

OUT VOID *Buffer

);
Parameters

- **This**: A pointer to the `EFI_SERIAL_IO_PROTOCOL` instance. Type `EFI_SERIAL_IO_PROTOCOL` is defined in Section 12.8.
- **BufferSize**: On input, the size of the `Buffer`. On output, the amount of data returned in `Buffer`.
- **Buffer**: The buffer to return the data into.

Description

The `Read()` function reads a specified number of bytes from a serial device. If a time out error or an overrun error is detected while data is being read from the serial device, then no more characters will be read, and an error will be returned. In all cases the number of bytes actually read is returned in `BufferSize`.

Status Codes Returned

<table>
<thead>
<tr>
<th>Code</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>EFI_SUCCESS</td>
<td>The data was read.</td>
</tr>
<tr>
<td>EFI_DEVICE_ERROR</td>
<td>The serial device reported an error.</td>
</tr>
<tr>
<td>EFI_TIMEOUT</td>
<td>The operation was stopped due to a timeout or overrun.</td>
</tr>
</tbody>
</table>

12.9 Graphics Output Protocol

The goal of this section is to replace the functionality that currently exists with VGA hardware and its corresponding video BIOS. The Graphics Output Protocol is a software abstraction and its goal is to support any foreseeable graphics hardware and not require VGA hardware, while at the same time also lending itself to implementation on the current generation of VGA hardware.

Graphics output is important in the pre-boot space to support modern firmware features. These features include the display of logos, the localization of output to any language, and setup and configuration screens.

Graphics output may also be required as part of the startup of an operating system. There are potentially times in modern operating systems prior to the loading of a high performance OS graphics driver where access to graphics output device is required. The Graphics Output Protocol supports this capability by providing the EFI OS loader access to a hardware frame buffer and enough information to allow the OS to draw directly to the graphics output device.

The `EFI_GRAPHICS_OUTPUT_PROTOCOL` supports three member functions to support the limited graphics needs of the pre-boot environment. These member functions allow the caller to draw to a virtualized frame buffer, retrieve the supported video modes, and to set a video mode. These simple primitives are sufficient to support the general needs of pre-OS firmware code.

The `EFI_GRAPHICS_OUTPUT_PROTOCOL` also exports enough information about the current mode for operating system startup software to access the linear frame buffer directly.

The interface structure for the Graphics Output protocol is defined in this section. A unique Graphics Output protocol must represent each video frame buffer in the system that is driven out to one or more video output devices.
12.9.1 Blt Buffer

The basic graphics operation in the EFI_GRAPHICS_OUTPUT_PROTOCOL is the Block Transfer or Blt. The Blt operation allows data to be read or written to the video adapter’s video memory. The Blt operation abstracts the video adapter’s hardware implementation by introducing the concept of a software Blt buffer.

The frame buffer abstracts the video display as an array of pixels. Each pixel’s location on the video display is defined by its X and Y coordinates. The X coordinate represents a scan line. A scan line is a horizontal line of pixels on the display. The Y coordinate represents a vertical line on the display. The upper left hand corner of the video display is defined as (0, 0) where the notation (X, Y) represents the X and Y coordinate of the pixel. The lower right corner of the video display is represented by (Width –1, Height -1).

The software Blt buffer is structured as an array of pixels. Pixel (0, 0) is the first element of the software Blt buffer. The Blt buffer can be thought of as a set of scan lines. It is possible to convert a pixel location on the video display to the Blt buffer using the following algorithm: Blt buffer array index = Y * Width + X.

Each software Blt buffer entry represents a pixel that is comprised of a 32-bit quantity. The color components of Blt buffer pixels are in PixelBlueGreenRedReserved8BitPerColor format as defined by EFI_GRAPHICS_OUTPUT_BLT_PIXEL. The byte values for the red, green, and blue components represent the color intensity. This color intensity value range from a minimum intensity of 0 to maximum intensity of 255.

![Software BLT Buffer](OM13157)
**EFI_GRAPHICS_OUTPUT_PROTOCOL**

**Summary**
Provides a basic abstraction to set video modes and copy pixels to and from the graphics controller’s frame buffer. The linear address of the hardware frame buffer is also exposed so software can write directly to the video hardware.

**GUID**
```c
#define EFI_GRAPHICS_OUTPUT_PROTOCOL_GUID
{0x9042a9de,0x23dc,0x4a38,}
{0x96,0xfb,0x7a,0xde,0xd0,0x80,0x51,0x6a}
```

**Protocol Interface Structure**
```c
typedef struct EFI_GRAPHICS_OUTPUT_PROTOCOL {
  EFI_GRAPHICS_OUTPUT_PROTOCOL_QUERY_MODE QueryMode;
  EFI_GRAPHICS_OUTPUT_PROTOCOL_SET_MODE SetMode;
  EFI_GRAPHICS_OUTPUT_PROTOCOL_BLT Blt;
  EFI_GRAPHICS_OUTPUT_PROTOCOL_MODE *Mode;
} EFI_GRAPHICS_OUTPUT_PROTOCOL;
```

**Parameters**
- **QueryMode**
  Returns information for an available graphics mode that the graphics device and the set of active video output devices supports.
- **SetMode**
  Set the video device into the specified mode and clears the visible portions of the output display to black.
- **Blt**
  Software abstraction to draw on the video device’s frame buffer.
- **Mode**
  Pointer to **EFI_GRAPHICS_OUTPUT_PROTOCOL_MODE** data. Type **EFI_GRAPHICS_OUTPUT_PROTOCOL_MODE** is defined in “Related Definitions” below.

**Related Definitions**
```c
typedef struct {
  UINT32  RedMask;
  UINT32  GreenMask;
  UINT32  BlueMask;
  UINT32  ReservedMask;
} EFI_PIXEL_BITMASK;
```

If a bit is set in **RedMask**, **GreenMask**, or **BlueMask** then those bits of the pixel represent the corresponding color. Bits in **RedMask**, **GreenMask**, **BlueMask**, and **ReservedMask** must not overlap bit positions. The values for the red, green, and blue components in the bit mask represent the color intensity. The color intensities must increase as the color values for each color mask increase with a minimum intensity of all bits in a color mask clear to a maximum intensity of all bits in a color mask set.
typedef enum {
    PixelRedGreenBlueReserved8BitPerColor,
    PixelBlueGreenRedReserved8BitPerColor,
    PixelBitMask,
    PixelBltOnly,
    PixelFormatMax
} EFI_GRAPHICS_PIXEL_FORMAT;

PixelRedGreenBlueReserved8BitPerColor
A pixel is 32-bits and byte zero represents red, byte one represents
green, byte two represents blue, and byte three is reserved. This is
the definition for the physical frame buffer. The byte values for the
red, green, and blue components represent the color intensity. This
color intensity value range from a minimum intensity of 0 to
maximum intensity of 255.

PixelBlueGreenRedReserved8BitPerColor
A pixel is 32-bits and byte zero represents blue, byte one represents
green, byte two represents red, and byte three is reserved. This is
the definition for the physical frame buffer. The byte values for the
red, green, and blue components represent the color intensity. This
color intensity value range from a minimum intensity of 0 to
maximum intensity of 255.

PixelBitMask
The pixel definition of the physical frame buffer is defined by
EFI PIXEL BITMASK.

PixelBltOnly
This mode does not support a physical frame buffer.

PixelFormatMax
Valid EFI GRAPHICS PIXEL FORMAT enum values are less than this
value.

typedef struct {
    UINT32                    Version;
    UINT32                    HorizontalResolution;
    UINT32                    VerticalResolution;
    EFI_GRAPHICS_PIXEL_FORMAT PixelFormat;
    EFI_PIXEL_BITMASK         PixelInformation;
    UINT32                    PixelsPerScanLine;
} EFI_GRAPHICS_OUTPUT_MODE_INFORMATION;

Version
The version of this data structure. A value of zero represents the
EFI GRAPHICS OUTPUT MODE INFORMATION structure as
defined in this specification. Future version of this specification may
extend this data structure in a backwards compatible way and
increase the value of Version.

HorizontalResolution
The size of video screen in pixels in the X dimension.

VerticalResolution
The size of video screen in pixels in the Y dimension.
PixelFormat

Enumeration that defines the physical format of the pixel. A value of `PixelBltOnly` implies that a linear frame buffer is not available for this mode.

PixelInformation

This bit-mask is only valid if `PixelFormat` is set to `PixelPixelBitMask`. A bit being set defines what bits are used for what purpose such as Red, Green, Blue, or Reserved.

PixelsPerScanLine

Defines the number of pixel elements per video memory line. For performance reasons, or due to hardware restrictions, scan lines may be padded to an amount of memory alignment. These padding pixel elements are outside the area covered by `HorizontalResolution` and are not visible. For direct frame buffer access, this number is used as a span between starts of pixel lines in video memory. Based on the size of an individual pixel element and `PixelsPerScanline`, the offset in video memory from pixel element (x, y) to pixel element (x, y+1) has to be calculated as "sizeof(PixelElement) * PixelsPerScanLine", not "sizeof(PixelElement) * HorizontalResolution", though in many cases those values can coincide. This value can depend on video hardware and mode resolution. GOP implementation is responsible for providing accurate value for this field.

Note: The following code sample is an example of the intended field usage:

```c
INTN GetPixelElementSize (IN EFI_PIXEL_BITMASK *PixelBits)
{
    INTN HighestPixel = -1;
    INTN BluePixel;
    INTN RedPixel;
    INTN GreenPixel;
    INTN RsvdPixel;

    BluePixel = FindHighestSetBit (PixelBits->BlueMask);
    RedPixel = FindHighestSetBit (PixelBits->RedMask);
    GreenPixel = FindHighestSetBit (PixelBits->GreenMask);
    RsvdPixel = FindHighestSetBit (PixelBits->ReservedMask);

    HighestPixel = max (BluePixel, RedPixel);
    HighestPixel = max (HighestPixel, GreenPixel);
    HighestPixel = max (HighestPixel, RsvdPixel);

    return HighestPixel;
}

EFI_PHYSICAL_ADDRESS NewPixelAddress;
```
EFI_PHYSICAL_ADDRESS CurrentPixelAddress;
EFI_GRAPHICS_OUTPUT_MODE_INFORMATION OutputInfo;
INTN PixelElementSize;

switch (OutputInfo.PixelFormat) {
  case PixelBitMask:
    PixelElementSize =
        GetPixelElementSize (&OutputInfo.PixelInformation);
    break;

  case PixelBlueGreenRedReserved8BitPerColor:
  case PixelRedGreenBlueReserved8BitPerColor:
    PixelElementSize =
        sizeof (EFI_GRAPHICS_OUTPUT_BLT_PIXEL);
    break;
}

//
// NewPixelAddress after execution points to the pixel
// positioned one line below the one pointed by
// CurrentPixelAddress
//
NewPixelAddress = CurrentPixelAddress +
  (PixelElementSize *
       OutputInfo.PixelsPerScanLine);

End of note code sample.

typedef struct {
  UINT32 MaxMode;
  UINT32 Mode;
  EFI_GRAPHICS_OUTPUT_MODE_INFORMATION *Info;
  UINTN SizeOfInfo;
  EFI_PHYSICAL_ADDRESS FrameBufferBase;
  UINTN FrameBufferSize;
} EFI_GRAPHICS_OUTPUT_PROTOCOL_MODE;

The **EFI_GRAPHICS_OUTPUT_PROTOCOL_MODE** is read-only and values are only changed by using the appropriate interface functions:

- **MaxMode**
  - The number of modes supported by `QueryMode()` and `SetMode()`.

- **Mode**
  - Current Mode of the graphics device. Valid mode numbers are 0 to `MaxMode` -1.

- **Info**
  - Pointer to read-only
    **EFI_GRAPHICS_OUTPUT_MODE_INFORMATION** data.
SizeOfInfo

Size of Info structure in bytes. Future versions of this specification may increase the size of the EFI_GRAPHICS_OUTPUT_MODE_INFORMATION data.

FrameBufferBase

Base address of graphics linear frame buffer. Info contains information required to allow software to draw directly to the frame buffer without using Blt(). Offset zero in FrameBufferBase represents the upper left pixel of the display.

FrameBufferSize

Amount of frame buffer needed to support the active mode as defined by PixelsPerScanLine x VerticalResolution x PixelElementSize.

Description

The EFI_GRAPHICS_OUTPUT_PROTOCOL provides a software abstraction to allow pixels to be drawn directly to the frame buffer. The EFI_GRAPHICS_OUTPUT_PROTOCOL is designed to be lightweight and to support the basic needs of graphics output prior to Operating System boot.

EFI_GRAPHICS_OUTPUT_PROTOCOL.QueryMode()

Summary

Returns information for an available graphics mode that the graphics device and the set of active video output devices supports.

Prototype

typedef
EFI_STATUS
EFI_GRAPHICS_OUTPUT_PROTOCOL_QUERY_MODE

(EFI_GRAPHICS_OUTPUT_PROTOCOL_QueryMode)

(IN EFI_GRAPHICS_OUTPUT_PROTOCOL *This,
IN UINT32 ModeNumber,
OUT UINTN *SizeOfInfo
OUT EFI_GRAPHICS_OUTPUT_MODE_INFORMATION **Info
);

Parameters

This

The EFI_GRAPHICS_OUTPUT_PROTOCOL instance. Type EFI_GRAPHICS_OUTPUT_PROTOCOL is defined in this section.

ModeNumber

The mode number to return information on. The current mode and valid modes are read-only values in the Mode structure of the EFI_GRAPHICS_OUTPUT_PROTOCOL.

SizeOfInfo

A pointer to the size, in bytes, of the Info buffer.

Info

A pointer to a callee allocated buffer that returns information about ModeNumber.

Description

The QueryMode() function returns information for an available graphics mode that the graphics device and the set of active video output devices supports. If ModeNumber is not between 0 and MaxMode – 1,
then **EFI_INVALID_PARAMETER** is returned. **MaxMode** is available from the **Mode** structure of the **EFI_GRAPHICS_OUTPUT_PROTOCOL**.

The size of the **Info** structure should never be assumed and the value of **SizeOfInfo** is the only valid way to know the size of **Info**.

If the **EFI_GRAPHICS_OUTPUT_PROTOCOL** is installed on the handle that represents a single video output device, then the set of modes returned by this service is the subset of modes supported by both the graphics controller and the video output device.

If the **EFI_GRAPHICS_OUTPUT_PROTOCOL** is installed on the handle that represents a combination of video output devices, then the set of modes returned by this service is the subset of modes supported by the graphics controller and the all of the video output devices represented by the handle.

### Status Codes Returned

<table>
<thead>
<tr>
<th>Code</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>EFI_SUCCESS</strong></td>
<td>Valid mode information was returned.</td>
</tr>
<tr>
<td><strong>EFI_DEVICE_ERROR</strong></td>
<td>A hardware error occurred trying to retrieve the video mode.</td>
</tr>
<tr>
<td><strong>EFI_INVALID_PARAMETER</strong></td>
<td><strong>ModeNumber</strong> is not valid.</td>
</tr>
</tbody>
</table>

### EFWI_GRAPHICS_OUTPUT_PROTOCOL.SetMode()

#### Summary

Set the video device into the specified mode and clears the visible portions of the output display to black.

#### Prototype

```c
typedef EFI_STATUS (EFIAPI *EFI_GRAPHICS_OUTPUT_PROTOCOL_SET_MODE) (
  IN EFI_GRAPHICS_OUTPUT_PROTOCOL *This,
  IN UINT32 ModeNumber
);
```

#### Parameters

- **This**: The **EFI_GRAPHICS_OUTPUT_PROTOCOL** instance. Type **EFI_GRAPHICS_OUTPUT_PROTOCOL** is defined in this section.
- **ModeNumber**: Abstraction that defines the current video mode. The current mode and valid modes are read-only values in the **Mode** structure of the **EFI_GRAPHICS_OUTPUT_PROTOCOL**.

#### Description

This **SetMode()** function sets the graphics device and the set of active video output devices to the video mode specified by **ModeNumber**. If **ModeNumber** is not supported **EFI_UNSUPPORTED** is returned.

If a device error occurs while attempting to set the video mode, then **EFI_DEVICE_ERROR** is returned. Otherwise, the graphics device is set to the requested geometry, the set of active output devices are set to the requested geometry, the visible portion of the hardware frame buffer is cleared to black, and **EFI_SUCCESS** is returned.
Status Codes Returned

<table>
<thead>
<tr>
<th>Code</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>EFI_SUCCESS</td>
<td>The graphics mode specified by <code>ModeNumber</code> was selected.</td>
</tr>
<tr>
<td>EFI_DEVICE_ERROR</td>
<td>The device had an error and could not complete the request.</td>
</tr>
<tr>
<td>EFI_UNSUPPORTED</td>
<td><code>ModeNumber</code> is not supported by this device.</td>
</tr>
</tbody>
</table>

EFI_GRAPHICS_OUTPUT_PROTOCOL.Blt()

Summary
Blt a rectangle of pixels on the graphics screen. Blt stands for BLock Transfer.

Prototype

```c
typedef struct {
    UINT8 Blue;
    UINT8 Green;
    UINT8 Red;
    UINT8 Reserved;
} EFI_GRAPHICS_OUTPUT_BLT_PIXEL;

typedef enum {
    EfiBltVideoFill,
    EfiBltVideoToBltBuffer,  
    EfiBltBufferToVideo,   
    EfiBltVideoToVideo, 
    EfiGraphicsOutputBltOperationMax 
} EFI_GRAPHICS_OUTPUT_BLT_OPERATION;

typedef EFI_STATUS (EFIAPI * EFI_GRAPHICS_OUTPUT_PROTOCOL_BLT) (
    IN EFI_GRAPHICS_OUTPUT_PROTOCOL *This,  
    IN OUT EFI_GRAPHICS_OUTPUT_BLT_PIXEL *BltBuffer, OPTIONAL  
    IN EFI_GRAPHICS_OUTPUT_BLT_OPERATION BltOperation,  
    IN UINTN SourceX,  
    IN UINTN SourceY, 
    IN UINTN DestinationX,  
    IN UINTN DestinationY, 
    IN UINTN Width,  
    IN UINTN Height,  
    IN UINTN Delta OPTIONAL  
);
```

Parameters

This The EFI_GRAPHICS_OUTPUT_PROTOCOL instance.
**BltBuffer**  
The data to transfer to the graphics screen. Size is at least  
Width*Height*sizeof(EFI_GRAPHICS_OUTPUT_BLT_PIXEL).

**BltOperation**  
The operation to perform when copying BltBuffer on to the graphics screen.

**SourceX**  
The X coordinate of the source for the BltOperation. The origin of the screen is 0, 0 and that is the upper left-hand corner of the screen.

**SourceY**  
The Y coordinate of the source for the BltOperation. The origin of the screen is 0, 0 and that is the upper left-hand corner of the screen.

**DestinationX**  
The X coordinate of the destination for the BltOperation. The origin of the screen is 0, 0 and that is the upper left-hand corner of the screen.

**DestinationY**  
The Y coordinate of the destination for the BltOperation. The origin of the screen is 0, 0 and that is the upper left-hand corner of the screen.

**Width**  
The width of a rectangle in the blt rectangle in pixels. Each pixel is represented by an EFI_GRAPHICS_OUTPUT_BLT_PIXEL element.

**Height**  
The height of a rectangle in the blt rectangle in pixels. Each pixel is represented by an EFI_GRAPHICS_OUTPUT_BLT_PIXEL element.

**Delta**  
Not used for EfiBltVideoFill or the EfiBltVideoToVideo operation. If a Delta of zero is used, the entire BltBuffer is being operated on. If a subrectangle of the BltBuffer is being used then Delta represents the number of bytes in a row of the BltBuffer.

**Description**

The Blt() function is used to draw the BltBuffer rectangle onto the video screen.

The BltBuffer represents a rectangle of Height by Width pixels that will be drawn on the graphics screen using the operation specified by BltOperation. The Delta value can be used to enable the BltOperation to be performed on a sub-rectangle of the BltBuffer.

Table 12-8 describes the BltOperations that are supported on rectangles. Rectangles have coordinates (left, upper) (right, bottom):

<table>
<thead>
<tr>
<th>Blt Operation</th>
<th>Operation</th>
</tr>
</thead>
<tbody>
<tr>
<td>EfiBltVideoFill</td>
<td>Write data from the BltBuffer pixel (0,0) directly to every pixel of the video display rectangle (DestinationX, DestinationY) (DestinationX + Width, DestinationY + Height). Only one pixel will be used from the BltBuffer. Delta is NOT used.</td>
</tr>
<tr>
<td>EfiBltVideoToBltBuffer</td>
<td>Read data from the video display rectangle (SourceX, SourceY) (SourceX + Width, SourceY + Height) and place it in the BltBuffer rectangle (DestinationX, DestinationY) (DestinationX + Width, DestinationY + Height). If DestinationX or DestinationY is not zero then Delta must be set to the length in bytes of a row in the BltBuffer.</td>
</tr>
</tbody>
</table>
EFI_EDID_DISCOVERED_PROTOCOL

Summary
This protocol contains the EDID information retrieved from a video output device.

GUID

```
#define EFI_EDID_DISCOVERED_PROTOCOL_GUID \
{0x1c0c34f6,0xd380,0x41fa,\ 
{0xa0,0x49,0x8a,0xd0,0x6c,0x1a,0x66,0xaa}}
```

Protocol Interface Structure

```
typedef struct {
  UINT32  SizeOfEdid;
  UINT8   *Edid;
} EFI_EDID_DISCOVERED_PROTOCOL;
```

Parameter

- **SizeOfEdid**
  The size, in bytes, of the `Edid` buffer. 0 if no EDID information is available from the video output device. Otherwise, it must be a minimum of 128 bytes.

- **Edid**
  A pointer to a read-only array of bytes that contains the EDID information for a video output device. This pointer is **NULL** if no EDID information is available from the video output device. The minimum size of a valid `Edid` buffer is 128 bytes. EDID information is defined in the E-EDID EEPROM specification published by VESA ([www.vesa.org](http://www.vesa.org)).
**Description**

EFI_EDID_DISCOVERED_PROTOCOL represents the EDID information that is returned from a video output device. If the video output device does not contain any EDID information, then the `SizeOfEdid` field must set to zero and the `Edid` field must be set to NULL. The EFI_EDID_DISCOVERED_PROTOCOL must be placed on every child handle that represents a possible video output device. The EFI_EDID_DISCOVERED_PROTOCOL is never placed on child handles that represent combinations of two or more video output devices.

**EFI_EDID_ACTIVE_PROTOCOL**

**Summary**

This protocol contains the EDID information for an active video output device. This is either the EDID information retrieved from the EFI_EDID_OVERRIDE_PROTOCOL if an override is available, or an identical copy of the EDID information from the EFI_EDID_DISCOVERED_PROTOCOL if no overrides are available.

**GUID**

```c
#define EFI_EDID_ACTIVE_PROTOCOL_GUID \
{0xbd8c1056,0x9f36,0x44ec,\ 
 {0x92,0xa8,0xa6,0x7f,0x81,0x79,0x86}}
```

**Protocol Interface Structure**

```c
typedef struct {
    UINT32  SizeOfEdid;
    UINT8  *Edid;
} EFI_EDID_ACTIVE_PROTOCOL;
```

**Parameter**

- `SizeOfEdid` The size, in bytes, of the `Edid` buffer. 0 if no EDID information is available from the video output device. Otherwise, it must be a minimum of 128 bytes.

- `Edid` A pointer to a read-only array of bytes that contains the EDID information for an active video output device. This pointer is NULL if no EDID information is available for the video output device. The minimum size of a valid `Edid` buffer is 128 bytes. EDID information is defined in the E-EDID EEPROM specification published by VESA (www.vesa.org).

**Description**

When the set of active video output devices attached to a frame buffer are selected, the EFI_EDID_ACTIVE_PROTOCOL must be installed onto the handles that represent the each of those active video output devices. If the EFI_EDID_OVERRIDE_PROTOCOL has override EDID information for an active video output device, then the EDID information specified by GetEdid() is used for the EFI_EDID_ACTIVE_PROTOCOL. Otherwise, the EDID information from the EFI_EDID_DISCOVERED_PROTOCOL is used for the EFI_EDID_ACTIVE_PROTOCOL. Since all EDID
information is read-only, it is legal for the pointer associated with the EFI_EDID_ACTIVE_PROTOCOL to
be the same as the pointer associated with the EFI_EDID_DISCOVERED_PROTOCOL when no overrides
are present.

**EFI_EDID_OVERRIDE_PROTOCOL**

**Summary**

This protocol is produced by the platform to allow the platform to provide EDID information to the
producer of the Graphics Output protocol.

**GUID**

```c
#define EFI_EDID_OVERRIDE_PROTOCOL_GUID
{0x48ecb431,0xfb72,0x45c0,
 {0xa9,0x22,0xf4,0x58,0xfe,0x04,0x0b,0xd5}}
```

**Protocol Interface Structure**

```c
typedef struct _EFI_EDID_OVERRIDE_PROTOCOL {
    EFI_EDID_OVERRIDE_PROTOCOL_GET_EDID GetEdid;
} EFI_EDID_OVERRIDE_PROTOCOL;
```

**Parameter**

- **GetEdid**

  Returns EDID values and attributes that the Video BIOS must use

**Description**

This protocol is produced by the platform to allow the platform to provide EDID information to the
producer of the Graphics Output protocol.

**EFI_EDID_OVERRIDE_PROTOCOL.GetEdid()**

**Summary**

Returns policy information and potentially a replacement EDID for the specified video output device.

**Prototype**

```c
typedef
 EFI_STATUS
 (EFIAPI *EFI_EDID_OVERRIDE_PROTOCOL_GET_EDID) (
     IN   EFI_EDID_OVERRIDE_PROTOCOL *This,
     IN   EFI_HANDLE *ChildHandle,
     OUT UINT32 *Attributes,
     OUT UINTN *EdidSize,
     OUT UINT8 **Edid
 );
```

**Parameters**

- **This**

  The EFI_EDID_OVERRIDE_PROTOCOL instance. Type

  EFI_EDID_OVERRIDE_PROTOCOL is defined in Section 12.10.
**ChildHandle**  A pointer to a child handle that represents a possible video output device.

**Attributes**  A pointer to the attributes associated with **ChildHandle** video output device.

**EdidSize**  A pointer to the size, in bytes, of the **Edid** buffer.

**Edid**  A pointer to the callee allocated buffer that contains the EDID information associated with **ChildHandle**. If **EdidSize** is 0, then a pointer to **NULL** is returned.

Note: the (EFI_HANDLE *) type of the "ChildHandle" parameter is an historical typing error in the UEFI specification. To match existent practice however, implementors and callers of the protocol are now expected to conform to the declaration of the parameter as written. That is, callers must pass the address of an EFI_HANDLE object as "ChildHandle", and implementors must dereference "ChildHandle" for finding the EFI_HANDLE object.

**Related Definitions**

```c
#define EFI_EDID_OVERRIDE_DONT_OVERRIDE  0x01
#define EFI_EDID_OVERRIDE_ENABLE_HOT_PLUG 0x02
```

**Table 12-9 Attributes Definition Table**

<table>
<thead>
<tr>
<th>Attribute Bit</th>
<th>EdidSize</th>
<th>Operation</th>
</tr>
</thead>
<tbody>
<tr>
<td>EFI_EDID_OVERRIDE_DONT_OVERRIDE=0</td>
<td>0</td>
<td>No override support for the display device.</td>
</tr>
<tr>
<td>EFI_EDID_OVERRIDE_DONT_OVERRIDE=0 != 0</td>
<td>!= 0</td>
<td>Always use returned override EDID for the display device.</td>
</tr>
<tr>
<td>EFI_EDID_OVERRIDE_DONT_OVERRIDE!=0</td>
<td>0</td>
<td>No override support for the display device.</td>
</tr>
<tr>
<td>EFI_EDID_OVERRIDE_DONT_OVERRIDE!=0 != 0</td>
<td>!= 0</td>
<td>Only use returned override EDID if the display device has no EDID or the EDID is incorrect. Otherwise, use the EDID from the display device.</td>
</tr>
<tr>
<td>EFI_EDID_OVERRIDE_ENABLE_HOT_PLUGIN=0</td>
<td>0</td>
<td>No hot plug support for the display device. A Graphics Output protocol will not be installed if no display device is not present.</td>
</tr>
<tr>
<td>EFI_EDID_OVERRIDE_ENABLE_HOT_PLUGIN!=0</td>
<td>!= 0</td>
<td>No hot plug support for the display device. The returned override EDID should be used according to the EFI_EDID_OVERRIDE_DONT_OVERRIDE attribute bit if the display device is present.</td>
</tr>
<tr>
<td>EFI_EDID_OVERRIDE_ENABLE_HOT_PLUGIN!=0</td>
<td>0</td>
<td>Invalid. The client of this protocol will not enable hot plug for the display device, and a Graphics Output protocol will not be installed if no other display is present.</td>
</tr>
</tbody>
</table>
Description

This protocol is optionally provided by the platform to override or provide EDID information and/or output device display properties to the producer of the Graphics Output protocol. If ChildHandle does not represent a video output device, or there are no override for the video output device specified by ChildHandle, then EFI_UNSUPPORTED is returned. Otherwise, the Attributes, EdidSize, and Edid parameters are returned along with a status of EFI_SUCCESS. Table 12-9 defines the behavior for the combinations of the Attribute and EdidSize parameters when EFI_SUCCESS is returned.

<table>
<thead>
<tr>
<th>Attribute Bit</th>
<th>EdidSize</th>
<th>Operation</th>
</tr>
</thead>
<tbody>
<tr>
<td>EFI_EDID_OVERRIDE_ENABLE_HOT_PLUG!=0</td>
<td>!= 0</td>
<td>Enable hot plug for the display device. A Graphics Output protocol will be installed even if the display device is not present.</td>
</tr>
</tbody>
</table>

Status Codes Returned

<table>
<thead>
<tr>
<th>Status Code</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>EFI_SUCCESS</td>
<td>Valid over ride returned for ChildHandle.</td>
</tr>
<tr>
<td>EFI_UNSUPPORTED</td>
<td>ChildHandle has no over rides.</td>
</tr>
</tbody>
</table>

12.10 Rules for PCI/AGP Devices

A UEFI driver that produces the Graphics Output Protocol must follow the UEFI driver model, produce an EFI_DRIVER_BINDING_PROTOCOL, and follow the rules on implementing the Supported(), Start(), and Stop(). The Start() function must not update the video output device in any way that is visible to the user. The Start() function must create child handle for each physical video output device and each supported combination of video output devices. The driver must retrieve the EDID information from each physical video output device and produce a EFI_EDID_DISCOVERED_PROTOCOL on the child handle that corresponds each physical video output device. The following summary describes the common initialization steps for a driver that produces the EFI_GRAPHICS_OUTPUT_PROTOCOL. This summary assumes the graphics controller supports a single frame buffer. If a graphics device supports multiple frame buffers, then handles for the frame buffers must be created first, and then the handles for the video output devices can be created as children of the frame buffer handles.

Summary of Initialization Steps:

- If RemainingDevicePath is NULL or the first Device Path Node is the End of Device Path Node, then Supported() returns EFI_SUCCESS. Otherwise, if the first node of RemainingDevicePath is not an ACPI_ADR node or the first two nodes of RemainingDevicePath are not a Controller node followed by an ACPI_ADR node, then Supported() returns EFI_UNSUPPORTED.
- If Supported() returned EFI_SUCCESS, system calls Start().
- If RemainingDevicePath is NULL, then a default set of active video output devices are selected by the driver.
- If the first Device Path Node of RemainingDevicePath is the End of Device Path Node, then skip to the "The EFI Driver must provide EFI_COMPONENT_NAME2_PROTOCOL" step.
• Start() function creates a ChildHandle for each physical video output device and installs the EFI_DEVICE_PATH_PROTOCOL onto the created ChildHandle. The EFI_DEVICE_PATH_PROTOCOL is constructed by appending an ACPI _ADR device path node describing the physical video output device to the end of the device path installed on the ControllerHandle passed into Start().

• Start() function retrieves EDID information for each physical video output device and installs the EFI_EDID_DISCOVERED_PROTOCOL onto the ChildHandle for each physical video output device. If no EDID data is available from the video output device, then SizeOfEdid is set to zero, and Edid is set to NULL.

• Start() function creates a ChildHandle for each valid combination of two or more video output devices, and installs the EFI_DEVICE_PATH_PROTOCOL onto the created ChildHandle. The EFI_DEVICE_PATH_PROTOCOL is constructed by appending an ACPI _ADR device path node describing the combination of video output devices to the end of the device path installed on the ControllerHandle passed into Start(). The ACPI _ADR entry can represent complex topologies of devices and it is possible to have more than one ACPI _ADR entry in a single device path node. Support of complex video output device topologies is an optional feature.

• Start() function evaluates the RemainingDevicePath to select the set of active video output devices. If RemainingDevicePath is NULL, then Start() selects a default set of video output devices. If RemainingDevicePath is not NULL, and ACPI _ADR device path node of RemainingDevicePath does not match any of the created ChildHandles, then Start() must destroy all its ChildHandles and return EFI_UNSUPPORTED. Otherwise, Start() selects the set of active video output devices specified by the ACPI _ADR device path node in RemainingDevicePath.

• Start() retrieves the ChildHandle associated with each active video output device. Only ChildHandles that represent a physical video output device are considered. Start() calls the EFI_EDID_OVERRIDE_PROTOCOL.GetEdid() service passing in ChildHandle. Depending on the return values from GetEdid(), either the override EDID information or the EDID information from the EFI_EDID_DISCOVERED_PROTOCOL on ChildHandle is selected. See GetEdid() for a detailed description of this decision. The selected EDID information is used to produce the EFI_EDID_ACTIVE_PROTOCOL, and that protocol is installed onto ChildHandle.

• Start() retrieves the one ChildHandle that represents the entire set of active video output devices. If this set is a single video output device, then this ChildHandle will be the same as the one used in the previous step. If this set is a combination of video output devices, then this will not be one of the ChildHandles used in the previous two steps. The EFI_GRAPHICS_OUTPUT_PROTOCOL is installed onto this ChildHandle.

• The QueryMode() service of the EFI_GRAPHICS_OUTPUT_PROTOCOL returns the set of modes that both the graphics controller and the set of active video output devices all support. If a different set of active video output device is selected, then a different set of modes will likely be produced by QueryMode().

• Start() function optionally initializes video frame buffer hardware. The EFI driver has the option of delaying this operation until SetMode() is called.

• The EFI Driver must provide EFI_COMPONENT_NAME2_PROTOCOL GetName() support for ControllerHandle and all the ChildHandles created by this driver. The name returned for ControllerHandle must return the name of the graphics device. The name
returned for each of the Child Handles allow the user to pick output display settings and should be constructed with this in mind.

- The EFI Driver’s Stop() function must cleanly undo what the Start() function created.
- An EFI_GRAPHICS_OUTPUT_PROTOCOL must be implemented for every video frame buffer that exists on a video adapter. In most cases there will be a single EFI_GRAPHICS_OUTPUT_PROTOCOL placed on one of the a children of the ControllerHandle passed into the EFI_DRIVER_BINDING.Start() function.

If a single PCI device/function contains multiple frame buffers the EFI_GRAPHICS_OUTPUT_PROTOCOL must create child handles of the PCI handle that inherit its PCI device path and appends a controller device path node. The handles for the video output devices are children of the handles that represent the frame buffers.

A video device can support an arbitrary number of geometries, but it must support one or more of the following modes to conform to this specification:

Onboard graphics device
- A mode required in a platform design guide
- Native mode of the display

Plug in graphics device
- A mode required in a platform design guide
- 800 x 600 with 32-bit color depth or 640 x 480 with 32-bit color depth and a pixel format described by PixelRedGreenBlueReserved8BitPerPixelColor or PixelBlueGreenRedReserved8BitPerPixelColor.

If graphics output device supports both landscape and portrait mode displays it must return a different mode via QueryMode(). For example landscape mode could be 800 horizontal and 600 vertical while the equivalent portrait mode would be 600 horizontal and 800 vertical.
13 - Protocols — Media Access

13.1 Load File Protocol

The Load File protocol is used to obtain files, that are primarily boot options, from arbitrary devices.

`EFI_LOAD_FILE_PROTOCOL`

**Summary**

Used to obtain files, that are primarily boot options, from arbitrary devices.

**GUID**

```c
#define EFI_LOAD_FILE_PROTOCOL_GUID  \
{0x56EC3091,0x954C,0x11d2,\  
{0x8e,0x3f,0x00,0xa0, 0xc9,0x69,0x72,0x3b}}
```

**Protocol Interface Structure**

```c
typedef struct _EFI_LOAD_FILE_PROTOCOL {
  EFI_LOAD_FILE     LoadFile;
} EFI_LOAD_FILE_PROTOCOL;
```

**Parameters**

- `LoadFile` Causes the driver to load the requested file. See the `LoadFile()` function description.

**Description**

The `EFI_LOAD_FILE_PROTOCOL` is a simple protocol used to obtain files from arbitrary devices.

When the firmware is attempting to load a file, it first attempts to use the device’s Simple File System protocol to read the file. If the file system protocol is found, the firmware implements the policy of interpreting the File Path value of the file being loaded. If the device does not support the file system protocol, the firmware then attempts to read the file via the `EFI_LOAD_FILE_PROTOCOL` and the `LoadFile()` function. In this case the `LoadFile()` function implements the policy of interpreting the File Path value.

`EFI_LOAD_FILE_PROTOCOL.LoadFile()`

**Summary**

Causes the driver to load a specified file.
Prototype

```c
typedef
EOI_STATUS
(EFIAP1 *EFI_LOAD_FILE) (  
    IN EFI_LOAD_FILE_PROTOCOL *This,
    IN EFI_DEVICE_PATH_PROTOCOL *FilePath,
    IN BOOLEAN BootPolicy,
    IN OUT UINTN *BufferSize,
    IN VOID *Buffer OPTIONAL
);
```

Parameters

- **This**
  Indicates a pointer to the calling context. Type `EFI_LOAD_FILE_PROTOCOL` is defined in Section 13.1.

- **FilePath**
  The device specific path of the file to load. Type `EFI_DEVICE_PATH_PROTOCOL` is defined in Section 10.2.

- **BootPolicy**
  If `TRUE`, indicates that the request originates from the boot manager, and that the boot manager is attempting to load `FilePath` as a boot selection. If `FALSE`, then `FilePath` must match an exact file to be loaded.

- **BufferSize**
  On input the size of `Buffer` in bytes. On output with a return code of `EFI_SUCCESS`, the amount of data transferred to `Buffer`. On output with a return code of `EFI_BUFFER_TOO_SMALL`, the size of `Buffer` required to retrieve the requested file.

- **Buffer**
  The memory buffer to transfer the file to. If `Buffer` is `NULL`, then the size of the requested file is returned in `BufferSize`. If `Buffer` is not `NULL`, and `BufferSize` is not large enough to hold the entire file, then `EFI_BUFFER_TOO_SMALL` is returned, and `BufferSize` is updated to indicate the size of the buffer needed to obtain the file. In this case, no data is returned in `Buffer`.

Description

The `LoadFile()` function interprets the device-specific `FilePath` parameter, returns the entire file into `Buffer`, and sets `BufferSize` to the amount of data returned. If `Buffer` is `NULL`, then the size of the file is returned in `BufferSize`. If `Buffer` is not `NULL`, and `BufferSize` is not large enough to hold the entire file, then `EFI_BUFFER_TOO_SMALL` is returned, and `BufferSize` is updated to indicate the size of the buffer needed to obtain the file. In this case, no data is returned in `Buffer`.

If `BootPolicy` is `FALSE` the `FilePath` must match an exact file to be loaded. If no such file exists, `EFI_NOT_FOUND` is returned. If `BootPolicy` is `FALSE`, and an attempt is being made to perform a network boot through the PXE Base Code protocol, `EFI_UNSUPPORTED` is returned.

If `BootPolicy` is `TRUE` the firmware’s boot manager is attempting to load an EFI image that is a boot selection. In this case, `FilePath` contains the file path value in the boot selection option. Normally the firmware would implement the policy on how to handle an inexact boot file path; however, since in this case the firmware cannot interpret the file path, the `LoadFile()` function is responsible for implementing the policy. For example, in the case of a network boot through the PXE Base Code protocol, `FilePath` merely points to the root of the device, and the firmware interprets this as wanting to boot from the first valid loader. The following is a list of events that `LoadFile()` will implement for a PXE boot:

- Perform DHCP.
• Optionally prompt the user with a menu of boot selections.
• Discover the boot server and the boot file.
• Download the boot file into Buffer and update BufferSize with the size of the boot file.

If the boot file downloaded from boot server is not an UEFI-formatted executable, but a binary image which contains a UEFI-compliant file system, then EFI_WARN_FILE_SYSTEM is returned, and a new RAM disk mapped on the returned Buffer is registered.

Status Codes Returned

<table>
<thead>
<tr>
<th>Code</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>EFI_SUCCESS</td>
<td>The file was loaded.</td>
</tr>
<tr>
<td>EFI_UNSUPPORTED</td>
<td>The device does not support the provided BootPolicy.</td>
</tr>
<tr>
<td>EFI_INVALID_PARAMETER</td>
<td>FilePath is not a valid device path, or BufferSize is NULL.</td>
</tr>
<tr>
<td>EFI_NO_MEDIA</td>
<td>No medium was present to load the file.</td>
</tr>
<tr>
<td>EFI_DEVICE_ERROR</td>
<td>The file was not loaded due to a device error.</td>
</tr>
<tr>
<td>EFI_NO_RESPONSE</td>
<td>The remote system did not respond.</td>
</tr>
<tr>
<td>EFI_NOT_FOUND</td>
<td>The file was not found.</td>
</tr>
<tr>
<td>EFI_ABORTED</td>
<td>The file load process was manually cancelled.</td>
</tr>
<tr>
<td>EFI_BUFFER_TOO_SMALL</td>
<td>The BufferSize is too small to read the current directory entry. BufferSize has been updated with the size needed to complete the request.</td>
</tr>
<tr>
<td>EFI_WARN_FILE_SYSTEM</td>
<td>The resulting Buffer contains UEFI-compliant file system.</td>
</tr>
</tbody>
</table>

13.2 Load File 2 Protocol

The Load File 2 protocol is used to obtain files from arbitrary devices that are not boot options.

**EFI_LOAD_FILE2_PROTOCOL**

**Summary**

Used to obtain files from arbitrary devices but are not used as boot options.

**GUID**

```c
#define EFI_LOAD_FILE2_PROTOCOL_GUID \
{ 0x4006c0c1, 0xfcb3, 0x403e, \ 
{ 0x99, 0x6d, 0x6a, 0x6c, 0x87, 0x24, 0xe0, 0x6d } }
```

**Protocol Interface Structure**

```c
typedef EFI_LOAD_FILE_PROTOCOL EFI_LOAD_FILE2_PROTOCOL;
```

**Parameters**

`LoadFile`

Causes the driver to load the requested file. See the LoadFile() functional description.
Description

The **EFI_LOAD_FILE2_PROTOCOL** is a simple protocol used to obtain files from arbitrary devices that are not boot options. It is used by **LoadImage()** when its **BootOption** parameter is **FALSE** and the **FilePath** does not have an instance of the **EFI_SIMPLE_FILE_SYSTEM_PROTOCOL**.

**EFI_LOAD_FILE2_PROTOCOL.LoadFile()**

**Summary**

Causes the driver to load a specified file.

**Prototype**

The same prototype as **EFI_LOAD_FILE_PROTOCOL.LoadFile()**.

**Parameters**

- **This**
  Indicates a pointer to the calling context.

- **FilePath**
  The device specific path of the file to load.

- **BootPolicy**
  Should always be **FALSE**.

- **BufferSize**
  On input the size of **Buffer** in bytes. On output with a return code of **EFI_SUCCESS**, the amount of data transferred to **Buffer**. On output with a return code of **EFI_BUFFER_TOO_SMALL**, the size of **Buffer** required to retrieve the requested file.

- **Buffer**
  The memory buffer to transfer the file to. If **Buffer** is **NULL**, then no the size of the requested file is returned in **BufferSize**.

**Description**

The **LoadFile()** function interprets the device-specific **FilePath** parameter, returns the entire file into **Buffer**, and sets **BufferSize** to the amount of data returned. If **Buffer** is **NULL**, then the size of the file is returned in **BufferSize**. If **Buffer** is not **NULL**, and **BufferSize** is not large enough to hold the entire file, then **EFI_BUFFER_TOO_SMALL** is returned, and **BufferSize** is updated to indicate the size of the buffer needed to obtain the file. In this case, no data is returned in **Buffer**.

**FilePath** contains the file path value in the boot selection option. Normally the firmware would implement the policy on how to handle an inexact boot file path; however, since in this case the firmware cannot interpret the file path, the **LoadFile()** function is responsible for implementing the policy.
Status Codes Returned

- **EFI_SUCCESS**: The file was loaded.
- **EFI_UNSUPPORTED**: BootPolicy is TRUE.
- **EFI_INVALID_PARAMETER**: FilePath is not a valid device path, or BufferSize is NULL.
- **EFI_NO_MEDIA**: No medium was present to load the file.
- **EFI_DEVICE_ERROR**: The file was not loaded due to a device error.
- **EFI_NO_RESPONSE**: The remote system did not respond.
- **EFI_NOT_FOUND**: The file was not found.
- **EFI_ABORTED**: The file load process was manually cancelled.
- **EFI_BUFFER_TOO_SMALL**: The BufferSize is too small to read the current directory entry. BufferSize has been updated with the size needed to complete the request.

### 13.3 File System Format

The file system supported by the Extensible Firmware Interface is based on the FAT file system. EFI defines a specific version of FAT that is explicitly documented and testable. Conformance to the EFI specification and its associate reference documents is the only definition of FAT that needs to be implemented to support EFI. To differentiate the EFI file system from pure FAT, a new partition file system type has been defined.

EFI encompasses the use of FAT32 for a system partition, and FAT12 or FAT16 for removable media. The FAT32 system partition is identified by an OSType value other than that used to identify previous versions of FAT. This unique partition type distinguishes an EFI defined file system from a normal FAT file system. The file system supported by EFI includes support for long file names.

The definition of the EFI file system will be maintained by specification and will not evolve over time to deal with errata or variant interpretations in OS file system drivers or file system utilities. Future enhancements and compatibility enhancements to FAT will not be automatically included in EFI file systems. The EFI file system is a target that is fixed by the EFI specification, and other specifications explicitly referenced by the EFI specification.

For more information about the EFI file system and file image format, visit the web site from which this document was obtained.

#### 13.3.1 System Partition

A System Partition is a partition in the conventional sense of a partition on a legacy system. For a hard disk, a partition is a contiguous grouping of sectors on the disk where the starting sector and size are defined by the Master Boot Record (MBR), which resides on LBA 0 (i.e., the first sector of the hard disk) (see Section 5.2), or the GUID Partition Table (GPT), which resides on logical block 1 (the second sector of the hard disk) (see Section 5.3.1). For a diskette (floppy) drive, a partition is defined to be the entire media. A System Partition can reside on any media that is supported by EFI Boot Services.

A System Partition supports backward compatibility with legacy systems by reserving the first block (sector) of the partition for compatibility code. On legacy systems, the first block (sector) of a partition is loaded into memory and execution is transferred to this code. EFI firmware does not execute the code in the MBR. The EFI firmware contains knowledge about the partition structure of various devices, and can understand legacy MBR, GPT, and “El Torito.”
The System Partition contains directories, data files, and UEFI Images. UEFI Images can contain a OS Loader, an driver to extend platform firmware capability, or an application that provides a transient service to the system. Applications written to this specification could include things such as a utility to create partitions or extended diagnostics. A System Partition can also support data files, such as error logs, that can be defined and used by various OS or system firmware software components.

13.3.1.1 File System Format

The first block (sector) of a partition contains a data structure called the BIOS Parameter Block (BPB) that defines the type and location of FAT file system on the drive. The BPB contains a data structure that defines the size of the media, the size of reserved space, the number of FAT tables, and the location and size of the root directory (not used in FAT32). The first block (sector) also contains code that will be executed as part of the boot process on a legacy system. This code in the first block (sector) usually contains code that can read a file from the root directory into memory and transfer control to it. Since EFI firmware contains a file system driver, EFI firmware can load any file from the file system with out needing to execute any code from the media.

The EFI firmware must support the FAT32, FAT16, and FAT12 variants of the EFI file system. What variant of EFI FAT to use is defined by the size of the media. The rules defining the relationship between media size and FAT variants is defined in the specification for the EFI file system.

The UEFI system partition FAT32 Data region should be aligned to the physical block boundary and optimal transfer length granularity of the device(see Section 5.3.1). This is controlled by the BPB_RsvdSecCnt field and the applicable BPB_FATSz field (e.g., formatting software may set the BPB_RsvdSecCnt field to a value that results in alignment and/or may set the BPB_FATSz field to a value that ensures alignment).

13.3.1.2 File Names

FAT stores file names in two formats. The original FAT format limited file names to eight characters with three extension characters. This type of file name is called an 8.3, pronounced eight dot three, file name. FAT was extended to include support for long file names (LFN).

FAT 8.3 file names are always stored as uppercase ASCII characters. LFN can either be stored as ASCII or UCS-2 characters and are stored case sensitive. The string that was used to open or create the file is stored directly into LFN. FAT defines that all files in a directory must have a unique name, and unique is defined as a case insensitive match. The following are examples of names that are considered to be the same and cannot exist in a single directory:

- “ThisIsAnExampleDirectory.Dir”
- “thisisanexamppledirectory.dir”
- THISISANEXAMPLEDIRECTORY.DIR
- ThisIsAnExampleDirectory.DIR
Note: Although the FAT32 specification allows file names to be encoded using UTF-16, this specification only recognizes the UCS-2 subset for the purposes of sorting or collation.

13.3.1.3 Directory Structure

An EFI system partition that is present on a hard disk must contain an EFI defined directory in the root directory. This directory is named EFI. All OS loaders and applications will be stored in subdirectories below EFI. Applications that are loaded by other applications or drivers are not required to be stored in any specific location in the EFI system partition. The choice of the subdirectory name is up to the vendor, but all vendors must pick names that do not collide with any other vendor’s subdirectory name. This applies to system manufacturers, operating system vendors, BIOS vendors, and third party tool vendors, or any other vendor that wishes to install files on an EFI system partition. There must also only be one executable EFI image for each supported processor architecture in each vendor subdirectory. This guarantees that there is only one image that can be loaded from a vendor subdirectory by the EFI Boot Manager. If more than one executable EFI image is present, then the boot behavior for the system will not be deterministic. There may also be an optional vendor subdirectory called BOOT.

This directory contains EFI images that aide in recovery if the boot selections for the software installed on the EFI system partition are ever lost. Any additional UEFI-compliant executables must be in subdirectories below the vendor subdirectory. The following is a sample directory structure for an EFI system partition present on a hard disk.

```
\EFI
  \<OS Vendor 1 Directory>
    <OS Loader Image>
  \<OS Vendor 2 Directory>
    <OS Loader Image>
  ...
  \<OS Vendor N Directory>
    <OS Loader Image>
  \<OEM Directory>
    <OEM Application Image>
  \<BIOS Vendor Directory>
    <BIOS Vendor Application Image>
  \<Third Party Tool Vendor Directory>
    <Third Party Tool Vendor Application Image>
\BOOT
  BOOT{machine type short name}.EFI
```

For removable media devices there must be only one UEFI-compliant system partition, and that partition must contain an UEFI-defined directory in the root directory. The directory will be named EFI. All OS loaders and applications will be stored in a subdirectory below EFI called BOOT. There must only be one executable EFI image for each supported processor architecture in the BOOT directory. For removable media to be bootable under EFI, it must be built in accordance with the rules laid out in Section 3.5.1.1. This guarantees that there is only one image that can be automatically loaded from a removable media device by the EFI Boot Manager. Any additional EFI executables must be in directories other than BOOT.

The following is a sample directory structure for an EFI system partition present on a removable media device.
13.3.2 Partition Discovery

This specification requires the firmware to be able to parse the legacy master boot record (MBR) (see Section 5.2.1), GUID Partition Table (GPT) (see Section 5.3.1), and El Torito (see Section 13.3.2.1) logical device volumes. The EFI firmware produces a logical EFI\_BLOCK\_IO\_PROTOCOL device for:

- each GUID Partition Entry (see table 16 in 5.3.3) with bit 1 set to zero;
- each El Torito logical device volume; and
- if no GPT is present, each partition found in the legacy MBR partition tables.

LBA zero of the EFI\_BLOCK\_IO\_PROTOCOL device will correspond to the first logical block of the partition. See Figure 13-1. If a GPT Partition Entry has Attribute bit 1 set then a logical EFI\_BLOCK\_IO\_PROTOCOL device must not be created.

---

Figure 13-1 Nesting of Legacy MBR Partition Records

The following is the order in which a block device must be scanned to determine if it contains partitions. When a check for a valid partitioning scheme succeeds, the search terminates.

1. Check for GUID Partition Table Headers.
2. Follow ISO-9660 specification to search for ISO-9660 volume structures on the magic LBA.
3. Check for an “El Torito” volume extension and follow the “El Torito” CD-ROM specification.
4. If none of the above, check LBA 0 for a legacy MBR partition table.
5. No partition found on device.
If a disk contains a recognized RAID structure (e.g. DDF structure as defined in The Storage Networking Industry Association Common RAID Disk Data Format Specification—see Glossary), the data on the disk must be ignored, unless the driver is using the RAID structure to produce a logical RAID volume.

EFI supports the nesting of legacy MBR partitions, by allowing any legacy MBR partition to contain more legacy MBR partitions. This is accomplished by supporting the same partition discovery algorithm on every logical block device. It should be noted that the GUID Partition Table does not allow nesting of GUID Partition Table Headers. Nesting is not needed since a GUID Partition Table Header can support an arbitrary number of partitions (the addressability limits of a 64-bit LBA are the limiting factor).

### 13.3.2.1 ISO-9660 and El Torito

ISO-9660 is the industry standard low level format used on CD-ROM and DVD-ROM. The CD-ROM format is completely described by the “El Torito” Bootable CD-ROM Format Specification Version 1.0. To boot from a CD-ROM or DVD-ROM in the boot services environment, an EFI System partition is stored in a “no emulation” mode as defined by the “El Torito” specification. A Platform ID of 0xEF indicates an EFI System Partition. The Platform ID is in either the Section Header Entry or the Validation Entry of the Booting Catalog as defined by the “El Torito” specification. EFI differs from “El Torito” “no emulation” mode in that it does not load the “no emulation” image into memory and jump to it. EFI interprets the “no emulation” image as an EFI system partition. EFI interprets the Sector Count in the Initial/Default Entry or the Section Header Entry to be the size of the EFI system partition. If the value of Sector Count is set to 0 or 1, EFI will assume the system partition consumes the space from the beginning of the “no emulation” image to the end of the CD-ROM.

A DVD-ROM image formatted as required by the UDF 2.0 specification (OSTA Universal Disk Format Specification, Revision 2.0) shall be booted by UEFI if:

- the DVD-ROM image conforms to the "UDF Bridge" format defined in the UDF 2.0 specification, and
- the DVD-ROM image contains exactly one ISO-9660 file system, and

Booting from a DVD-ROM that satisfies the above requirements is accomplished using the same methods as booting from a CD-ROM: the ISO-9660 file system shall be booted.

Since the EFI file system definition does not use the same Initial/Default entry as a legacy CD-ROM it is possible to boot personal computers using an EFI CD-ROM or DVD-ROM. The inclusion of boot code for personal computers is optional and not required by EFI.

### 13.3.3 Number and Location of System Partitions

UEFI does not impose a restriction on the number or location of System Partitions that can exist on a system. System Partitions are discovered when required by UEFI firmware by examining the partition GUID and verifying that the contents of the partition conform to the FAT file system as defined in Section 13.3.1.1. Further, UEFI implementations may allow the use of conforming FAT partitions which do not use the ESP GUID. Partition creators may prevent UEFI firmware from examining and using a specific partition by setting bit 1 of the Partition Attributes (see 5.3.3) which will exclude the partition as a potential ESP.
Software installation may choose to create and locate an ESP on each target OS boot disk, or may choose to create a single ESP independent of the location of OS boot disks and OS partitions. It is outside of the scope of this specification to attempt to coordinate the specification of size and location of an ESP that can be shared by multiple OS or Diagnostics installations, or to manage potential namespace collisions in directory naming in a single (central) ESP.

### 13.3.4 Media Formats

This section describes how booting from different types of removable media is handled. In general the rules are consistent regardless of a media’s physical type and whether it is removable or not.

#### 13.3.4.1 Removable Media

Removable media may contain a standard FAT12, FAT16, or FAT32 file system.

Booting from a removable media device can be accomplished the same way as any other boot. The boot file path provided to the boot manager can consist of a UEFI application image to load, or can merely be the path to a removable media device. In the first case, the path clearly indicates the image that is to be loaded. In the later case, the boot manager implements the policy to load the default application image from the device.

For removable media to be bootable under EFI, it must be built in accordance with the rules laid out in Section 3.5.1.1.

#### 13.3.4.2 Diskette

EFI bootable diskettes follow the standard formatting conventions used on personal computers. The diskette contains only a single partition that complies to the EFI file system type. For diskettes to be bootable under EFI, it must be built in accordance with the rules laid out in Section 3.5.1.1.

Since the EFI file system definition does not use the code in the first block of the diskette, it is possible to boot personal computers using a diskette that is also formatted as an EFI bootable removable media device. The inclusion of boot code for personal computers is optional and not required by EFI.

Diskettes include the legacy 3.5-inch diskette drives as well as the newer larger capacity removable media drives such as an Iomega Zip*, Fujitsu MO, or MKE LS-120/SuperDisk*.

#### 13.3.4.3 Hard Drive

Hard drives may contain multiple partitions as defined in Section 13.3.2 on partition discovery. Any partition on the hard drive may contain a file system that the EFI firmware recognizes. Images that are to be booted must be stored under the EFI subdirectory as defined in Section 13.3.1 and Section 13.3.2.

EFI code does not assume a fixed block size.

Since EFI firmware does not execute the MBR code and does not depend on the BootIndicator field in the legacy MBR partition records, the hard disk can still boot and function normally.

#### 13.3.4.4 CD-ROM and DVD-ROM

A CD-ROM or DVD-ROM may contain multiple partitions as defined Section 13.3.1 and Section 13.3.2 and in the “El Torito” specification.
EFI code does not assume a fixed block size.

Since the EFI file system definition does not use the same Initial/Default entry as a legacy CD-ROM, it is possible to boot personal computers using an EFI CD-ROM or DVD-ROM. The inclusion of boot code for personal computers is optional and not required by EFI.

### 13.3.4.5 Network

To boot from a network device, the Boot Manager uses the Load File Protocol to perform a `LoadFile()` on the network device. This uses the PXE Base Code Protocol to perform DHCP and Discovery. This may result in a list of possible boot servers along with the boot files available on each server. The Load File Protocol for a network boot may then optionally produce a menu of these selections for the user to choose from. If this menu is presented, it will always have a timeout, so the Load File Protocol can automatically boot the default boot selection. If there is only one possible boot file, then the Load File Protocol can automatically attempt to load the one boot file.

The Load File Protocol will download the boot file using the MTFTP service in the PXE Base Code Protocol. The downloaded image must be an EFI image that the platform supports.

### 13.4 Simple File System Protocol

The Simple File System protocol allows code running in the EFI boot services environment to obtain file based access to a device. `EFI_SIMPLE_FILE_SYSTEM_PROTOCOL` is used to open a device volume and return an `EFI_FILE_PROTOCOL` that provides interfaces to access files on a device volume.

#### EFI_SIMPLE_FILE_SYSTEM_PROTOCOL

**Summary**

Provides a minimal interface for file-type access to a device.

**GUID**

```c
#define EFI_SIMPLE_FILE_SYSTEM_PROTOCOL_GUID {0x0964e5b22,0x6459,0x11d2,0x8e,0x39,0x00,0xa0,0xc9,0x72,0x3b}
```

**Revision Number**

```c
#define EFI_SIMPLE_FILE_SYSTEM_PROTOCOL_REVISION 0x00010000
```

**Protocol Interface Structure**

```c
typedef struct _EFI_SIMPLE_FILE_SYSTEM_PROTOCOL {
    UINT64 Revision;
    EFI_SIMPLE_FILE_SYSTEM_PROTOCOL_OPEN_VOLUME OpenVolume;
} EFI_SIMPLE_FILE_SYSTEM_PROTOCOL;
```

**Parameters**

- **Revision**
  
  The version of the `EFI_FILE_PROTOCOL`. The version specified by this specification is 0x00010000. All future revisions must be backwards
compatible. If a future version is not backwards compatible, it is not the same GUID.

OpenVolume

Opens the volume for file I/O access. See the OpenVolume() function description.

Description

The EFI_SIMPLE_FILE_SYSTEM_PROTOCOL provides a minimal interface for file-type access to a device. This protocol is only supported on some devices.

Devices that support the Simple File System protocol return an EFI_FILE_PROTOCOL. The only function of this interface is to open a handle to the root directory of the file system on the volume. Once opened, all accesses to the volume are performed through the volume’s file handles, using the EFI_FILE_PROTOCOL protocol. The volume is closed by closing all the open file handles.

The firmware automatically creates handles for any block device that supports the following file system formats:

- FAT12
- FAT16
- FAT32

EFI_SIMPLE_FILE_SYSTEM_PROTOCOL.OpenVolume()

Summary

Opens the root directory on a volume.

Prototype

typedef

EFI_STATUS

(EIFIAPI *EFI_SIMPLE_FILE_SYSTEM_PROTOCOL_OPEN_VOLUME) (  
  IN EFI_SIMPLE_FILE_SYSTEM_PROTOCOL *This,  
  OUT EFI_FILE_PROTOCOL **Root

);

Parameters

This

A pointer to the volume to open the root directory of. See the type EFI_SIMPLE_FILE_SYSTEM_PROTOCOL description.

Root

A pointer to the location to return the opened file handle for the root directory. See the type EFI_FILE_PROTOCOL description.

Description

The OpenVolume() function opens a volume, and returns a file handle to the volume’s root directory. This handle is used to perform all other file I/O operations. The volume remains open until all the file handles to it are closed.

If the medium is changed while there are open file handles to the volume, all file handles to the volume will return EFI_MEDIA_CHANGED. To access the files on the new medium, the volume must be reopened with OpenVolume(). If the new medium is a different file system than the one supplied in the
**EFI_HANDLE**’s *DevicePath* for the **EFI_SIMPLE_SYSTEM_PROTOCOL**, **OpenVolume()** will return **EFI_UNSUPPORTED**.

### Status Codes Returned

<table>
<thead>
<tr>
<th>Code</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>EFI_SUCCESS</td>
<td>The file volume was opened.</td>
</tr>
<tr>
<td>EFI_UNSUPPORTED</td>
<td>The volume does not support the requested file system type.</td>
</tr>
<tr>
<td>EFI_NO_MEDIA</td>
<td>The device has no medium.</td>
</tr>
<tr>
<td>EFI_DEVICE_ERROR</td>
<td>The device reported an error.</td>
</tr>
<tr>
<td>EFI_VOLUME_CORRUPTED</td>
<td>The file system structures are corrupted.</td>
</tr>
<tr>
<td>EFI_ACCESS_DENIED</td>
<td>The service denied access to the file.</td>
</tr>
<tr>
<td>EFI_OUT_OF_RESOURCES</td>
<td>The file volume was not opened.</td>
</tr>
<tr>
<td>EFI_MEDIA_CHANGED</td>
<td>The device has a different medium in it or the medium is no longer supported. To access the files on the new medium, the volume must be reopened with <strong>OpenVolume()</strong>.</td>
</tr>
</tbody>
</table>

### 13.5 File Protocol

The protocol and functions described in this section support access to **EFI-supported file systems**.

**EFI_FILE_PROTOCOL**

**Summary**

Provides file based access to supported file systems.

**Revision Number**

```
#define EFI_FILE_PROTOCOL_REVISION    0x00010000
#define EFI_FILE_PROTOCOL_REVISION2   0x00020000
#define EFI_FILE_PROTOCOL_LATEST_REVISION EFI_FILE_PROTOCOL_REVISION2
```
Protocol Interface Structure

```c
typedef struct _EFI_FILE_PROTOCOL {
    UINT64 Revision;
    EFI_FILE_OPEN Open;
    EFI_FILE_CLOSE Close;
    EFI_FILE_DELETE Delete;
    EFI_FILE_READ Read;
    EFI_FILE_WRITE Write;
    EFI_FILE_GET_POSITIONGetPosition;
    EFI_FILE_SET_POSITIONSetPosition;
    EFI_FILE_GET_INFOGetInfo;
    EFI_FILE_SET_INFOSetInfo;
    EFI_FILE_FLUSHFlush;
    EFI_FILE_OPEN_EX OpenEx; // Added for revision 2
    EFI_FILE_READ_EX ReadEx; // Added for revision 2
    EFI_FILE_WRITE_EX WriteEx; // Added for revision 2
    EFI_FILE_FLUSH_EX FlushEx; // Added for revision 2
} EFI_FILE_PROTOCOL;
```

Parameters

- **Revision**
  The version of the `EFI_FILE_PROTOCOL` interface. The version specified by this specification is `EFI_FILE_PROTOCOL_LATEST_REVISION`. Future versions are required to be backward compatible to version 1.0.

- **Open**
  Opens or creates a new file. See the `Open()` function description.

- **Close**
  Closes the current file handle. See the `Close()` function description.

- **Delete**
  Deletes a file. See the `Delete()` function description.

- **Read**
  Reads bytes from a file. See the `Read()` function description.

- **Write**
  Writes bytes to a file. See the `Write()` function description.

- **GetPosition**
  Returns the current file position. See the `GetPosition()` function description.

- **SetPosition**
  Sets the current file position. See the `SetPosition()` function description.

- **GetInfo**
  Gets the requested file or volume information. See the `GetInfo()` function description.

- **SetInfo**
  Sets the requested file information. See the `SetInfo()` function description.

- **Flush**
  Flushes all modified data associated with the file to the device. See the `Flush()` function description.

- **OpenEx**
  Opens a new file relative to the source directory’s location.

- **ReadEx**
  Reads data from a file.

- **WriteEx**
  Writes data to a file.
FlushEx

Flushes all modified data associated with a file to a device.

Description

The EFI_FILE_PROTOCOL provides file I/O access to supported file systems.

An EFI_FILE_PROTOCOL provides access to a file’s or directory’s contents, and is also a reference to a location in the directory tree of the file system in which the file resides. With any given file handle, other files may be opened relative to this file’s location, yielding new file handles.

On requesting the file system protocol on a device, the caller gets the EFI_FILE_PROTOCOL to the volume. This interface is used to open the root directory of the file system when needed. The caller must Close() the file handle to the root directory, and any other opened file handles before exiting. While there are open files on the device, usage of underlying device protocol(s) that the file system is abstracting must be avoided. For example, when a file system that is layered on a EFI_DISK_IO_PROTOCOL/EFI_BLOCK_IO_PROTOCOL, direct block access to the device for the blocks that comprise the file system must be avoided while there are open file handles to the same device.

A file system driver may cache data relating to an open file. A Flush() function is provided that flushes all dirty data in the file system, relative to the requested file, to the physical medium. If the underlying device may cache data, the file system must inform the device to flush as well.

Implementations must account for cases where there is pending queued asynchronous I/O when a call is received on a blocking protocol interface. In these cases the pending I/O will be processed and completed before the blocking function is executed so that operation are carried out in the order they were requested.

EFI_FILE_PROTOCOL.Open()

Summary

Opens a new file relative to the source file’s location.

Prototype

typedef EFI_STATUS (EFIAPI *EFI_FILE_OPEN) ( 
    IN EFI_FILE_PROTOCOL *This, 
    OUT EFI_FILE_PROTOCOL **NewHandle, 
    IN CHAR16 *FileName, 
    IN UINT64 OpenMode, 
    IN UINT64 Attributes 
); 

Parameters

This

A pointer to the EFI_FILE_PROTOCOL instance that is the file handle to the source location. This would typically be an open handle to a directory. See the type EFI_FILE_PROTOCOL description.

NewHandle

A pointer to the location to return the opened handle for the new file. See the type EFI_FILE_PROTOCOL description.
**FileName**
The Null-terminated string of the name of the file to be opened. The file name may contain the following path modifiers: \\
, ".", and "..".

**OpenMode**
The mode to open the file. The only valid combinations that the file may be opened with are: Read, Read/Write, or Create/Read/Write. See “Related Definitions” below.

**Attributes**
Only valid for **EFI_FILE_MODE_CREATE**, in which case these are the attribute bits for the newly created file. See “Related Definitions” below.

### Related Definitions
```c
//*******************************************************
// Open Modes
//*******************************************************
#define EFI_FILE_MODE_READ     0x0000000000000001
#define EFI_FILE_MODE_WRITE    0x0000000000000002
#define EFI_FILE_MODE_CREATE   0x8000000000000000
```

```c
//*******************************************************
// File Attributes
//*******************************************************
#define EFI_FILE_READ_ONLY     0x0000000000000001
#define EFI_FILE_HIDDEN        0x0000000000000002
#define EFI_FILE_SYSTEM        0x0000000000000004
#define EFI_FILE_RESERVED      0x0000000000000008
#define EFI_FILE_DIRECTORY     0x0000000000000010
#define EFI_FILE_ARCHIVE       0x0000000000000020
#define EFI_FILE_VALID_ATTR    0x0000000000000037
```

### Description
The **Open()** function opens the file or directory referred to by **FileName** relative to the location of **This** and returns a **NewHandle**. The **FileName** may include the following path modifiers:

```
\"\" If the filename starts with a \"\" the relative location is the root directory that **This** resides on; otherwise \"\" separates name components. Each name component is opened in turn, and the handle to the last file opened is returned.

\"\" Opens the current location.

\..\" Opens the parent directory for the current location. If the location is the root directory the request will return an error, as there is no parent directory for the root directory.
```

If **EFI_FILE_MODE_CREATE** is set, then the file is created in the directory. If the final location of **FileName** does not refer to a directory, then the operation fails. If the file does not exist in the directory, then a new file is created. If the file already exists in the directory, then a new file is created.

If the medium of the device changes, all accesses (including the File handle) will result in **EFI_MEDIA_CHANGED**. To access the new medium, the volume must be reopened.
Status Codes Returned

<table>
<thead>
<tr>
<th>Status Code</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>EFI_SUCCESS</td>
<td>The file was opened.</td>
</tr>
<tr>
<td>EFI_NOT_FOUND</td>
<td>The specified file could not be found on the device.</td>
</tr>
<tr>
<td>EFI_NO_MEDIA</td>
<td>The device has no medium.</td>
</tr>
<tr>
<td>EFI_MEDIA_CHANGED</td>
<td>The device has a different medium in it or the medium is no longer supported.</td>
</tr>
<tr>
<td>EFI_DEVICE_ERROR</td>
<td>The device reported an error.</td>
</tr>
<tr>
<td>EFI_VOLUME_CORRUPTED</td>
<td>The file system structures are corrupted.</td>
</tr>
<tr>
<td>EFI_WRITE_PROTECTED</td>
<td>An attempt was made to create a file, or open a file for write when the media is write-protected.</td>
</tr>
<tr>
<td>EFI_ACCESS_DENIED</td>
<td>The service denied access to the file.</td>
</tr>
<tr>
<td>EFI_OUT_OF_RESOURCES</td>
<td>Not enough resources were available to open the file.</td>
</tr>
<tr>
<td>EFI_VOLUME_FULL</td>
<td>The volume is full.</td>
</tr>
</tbody>
</table>

**EFI_FILE_PROTOCOL.Close()**

**Summary**
Closes a specified file handle.

**Prototype**
```c
typedef EFI_STATUS
    (EFIAPI *EFI_FILE_CLOSE) (
    IN EFI_FILE_PROTOCOL  *This
    );
```

**Parameters**
*This* A pointer to the EFI_FILE_PROTOCOL instance that is the file handle to close. See the type EFI_FILE_PROTOCOL description.

**Description**
The Close() function closes a specified file handle. All “dirty” cached file data is flushed to the device, and the file is closed. In all cases the handle is closed. The operation will wait for all pending asynchronous I/O requests to complete before completing.

**Status Codes Returned**

<table>
<thead>
<tr>
<th>Status Code</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>EFI_SUCCESS</td>
<td>The file was closed.</td>
</tr>
</tbody>
</table>

**EFI_FILE_PROTOCOL.Delete()**

**Summary**
Closes and deletes a file.
Prototype

typedef
    EFI_STATUS
    (EFIAPI *EFI_FILE_DELETE) (  
        IN EFI_FILE_PROTOCOL  *This
    );

Parameters

This

A pointer to the EFI_FILE_PROTOCOL instance that is the handle to the file to delete. See the type EFI_FILE_PROTOCOL description.

Description

The Delete() function closes and deletes a file. In all cases the file handle is closed. If the file cannot be deleted, the warning code EFI_WARN_DELETE_FAILURE is returned, but the handle is still closed.

Status Codes Returned

<table>
<thead>
<tr>
<th>Code</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>EFI_SUCCESS</td>
<td>The file was closed and deleted, and the handle was closed.</td>
</tr>
<tr>
<td>EFI_WARN_DELETE_FAILURE</td>
<td>The handle was closed, but the file was not deleted.</td>
</tr>
</tbody>
</table>

EFI_FILE_PROTOCOL.Read()

Summary

Reads data from a file.

Prototype

typedef
    EFI_STATUS
    (EFIAPI *EFI_FILE_READ) (  
        IN EFI_FILE_PROTOCOL  *This,
        IN OUT UINTN        *BufferSize,
        OUT VOID           *Buffer
    );

Parameters

This

A pointer to the EFI_FILE_PROTOCOL instance that is the file handle to read data from. See the type EFI_FILE_PROTOCOL description.

BufferSize

On input, the size of the Buffer. On output, the amount of data returned in Buffer. In both cases, the size is measured in bytes.

Buffer

The buffer into which the data is read.

Description

The Read() function reads data from a file.
If `This` is not a directory, the function reads the requested number of bytes from the file at the file’s current position and returns them in `Buffer`. If the read goes beyond the end of the file, the read length is truncated to the end of the file. The file’s current position is increased by the number of bytes returned.

If `This` is a directory, the function reads the directory entry at the file’s current position and returns the entry in `Buffer`. If the `Buffer` is not large enough to hold the current directory entry, then `EFI_BUFFER_TOO_SMALL` is returned and the current file position is not updated. `BufferSize` is set to be the size of the buffer needed to read the entry. On success, the current position is updated to the next directory entry. If there are no more directory entries, the read returns a zero-length buffer. `EFI_FILE_INFO` is the structure returned as the directory entry.

### Status Codes Returned

<table>
<thead>
<tr>
<th>Code</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>EFI_SUCCESS</td>
<td>The data was read.</td>
</tr>
<tr>
<td>EFI_NO_MEDIA</td>
<td>The device has no medium.</td>
</tr>
<tr>
<td>EFI_DEVICE_ERROR</td>
<td>The device reported an error.</td>
</tr>
<tr>
<td>EFI_DEVICE_ERROR</td>
<td>An attempt was made to read from a deleted file.</td>
</tr>
<tr>
<td>EFI_DEVICE_ERROR</td>
<td>On entry, the current file position is beyond the end of the file.</td>
</tr>
<tr>
<td>EFI_VOLUME_CORRUPTED</td>
<td>The file system structures are corrupted.</td>
</tr>
<tr>
<td>EFI_BUFFER_TOO_SMALL</td>
<td>The <code>BufferSize</code> is too small to read the current directory entry. The <code>BufferSize</code> has been updated with the size needed to complete the request.</td>
</tr>
</tbody>
</table>

###EFI_FILE_PROTOCOL.Write()

**Summary**

Writes data to a file.

**Prototype**

```c
typedef EFI_STATUS (EFIAPI *EFI_FILE_WRITE) ( 
    IN EFI_FILE_PROTOCOL   *This, 
    IN OUT UINTN           *BufferSize, 
    IN VOID                *Buffer 
);
```

**Parameters**

- `This` A pointer to the EFI_FILE_PROTOCOL instance that is the file handle to write data to. See the type EFI_FILE_PROTOCOL description.
- `BufferSize` On input, the size of the `Buffer`. On output, the amount of data actually written. In both cases, the size is measured in bytes.
- `Buffer` The buffer of data to write.
**Description**

The `Write()` function writes the specified number of bytes to the file at the current file position. The current file position is advanced the actual number of bytes written, which is returned in `BufferSize`. Partial writes only occur when there has been a data error during the write attempt (such as “file space full”). The file is automatically grown to hold the data if required.

Direct writes to opened directories are not supported.

**Status Codes Returned**

<table>
<thead>
<tr>
<th>Status Code</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>EFI_SUCCESS</td>
<td>The data was written.</td>
</tr>
<tr>
<td>EFI_UNSUPPORT</td>
<td>Writes to open directory files are not supported.</td>
</tr>
<tr>
<td>EFI_NO_MEDIA</td>
<td>The device has no medium.</td>
</tr>
<tr>
<td>EFI_DEVICE_ERROR</td>
<td>The device reported an error.</td>
</tr>
<tr>
<td>EFI_DEVICE_ERROR</td>
<td>An attempt was made to write to a deleted file.</td>
</tr>
<tr>
<td>EFI_VOLUME_CORRUPTED</td>
<td>The file system structures are corrupted.</td>
</tr>
<tr>
<td>EFI_WRITE_PROTECTED</td>
<td>The file or medium is write-protected.</td>
</tr>
<tr>
<td>EFI_ACCESS_DENIED</td>
<td>The file was opened read only.</td>
</tr>
<tr>
<td>EFI_VOLUME_FULL</td>
<td>The volume is full.</td>
</tr>
</tbody>
</table>

**EFI_FILE_PROTOCOL.OpenEx()**

**Summary**

Opens a new file relative to the source directory’s location.

**Prototype**

```c
typedef
    EFI_STATUS
    (EFIAPI *EFI_FILE_OPEN_EX) (  
        IN EFI_FILE_PROTOCOL *This,  
        OUT EFI_FILE_PROTOCOL **NewHandle,  
        IN CHAR16 *FileName,  
        IN UINT64 OpenMode,  
        IN UINT64 Attributes,  
        IN OUT EFI_FILE_IO_TOKEN *Token  
    );
```

**Parameters**

- **This**
  A pointer to the `EFI_FILE_PROTOCOL` instance that is the file handle to read data from. See the type `EFI_FILE_PROTOCOL` description.

- **NewHandle**
  A pointer to the location to return the opened handle for the new file. See the type `EFI_FILE_PROTOCOL` description. For asynchronous I/O, this pointer must remain valid for the duration of the asynchronous operation.
**FileName**  
The Null-terminated string of the name of the file to be opened. The file name may contain the following path modifiers: "\", ",", and ".".

**OpenMode**  
The mode to open the file. The only valid combinations that the file may be opened with are: Read, Read/Write, or Create/Read/Write. See “Related Definitions” below.

**Attributes**  
Only valid for **EFI_FILE_MODE_CREATE**, in which case these are the attribute bits for the

**Token**  
A pointer to the token associated with the transaction. Type  
**EFI_FILE_IO_TOKEN** is defined in "Related Definitions" below.

**Description**  
The **OpenEx()** function opens the file or directory referred to by **FileName** relative to the location of **This** and returns a **NewHandle**. The **FileName** may include the path modifiers described previously in **Open()**.

If **EFI_FILE_MODE_CREATE** is set, then the file is created in the directory. If the final location of **FileName** does not refer to a directory, then the operation fails. If the file does not exist in the directory, then a new file is created. If the file already exists in the directory, then the existing file is opened.

If the medium of the device changes, all accesses (including the File handle) will result in **EFI_MEDIA_CHANGED**. To access the new medium, the volume must be reopened.

If an error is returned from the call to **OpenEx()** and non-blocking I/O is being requested, the **Event** associated with this request will not be signaled. If the call to **OpenEx()** succeeds then the **Event** will be signaled upon completion of the open or if an error occurs during the processing of the request. The status of the read request can be determined from the Status field of the Token once the event is signaled.

**Related Definitions**

typedef struct {
  EFI_EVENT Event;
  EFI_STATUS Status;
  UINTN BufferSize;
  VOID *Buffer;
} EFI_FILE_IO_TOKEN;

**Event**  
If **Event** is NULL, then blocking I/O is performed. If **Event** is not NULL and non-blocking I/O is supported, then non-blocking I/O is performed, and **Event** will be signaled when the read request is completed. The caller must be prepared to handle the case where the callback associated with **Event** occurs before the original asynchronous I/O request call returns.

**Status**  
Defines whether or not the signaled event encountered an error.

**BufferSize**  
For **OpenEx()**: Not Used, ignored  
For **ReadEx()**: On input, the size of the **Buffer**. On output, the amount of data returned in **Buffer**. In both cases, the size is measured in bytes.
For **WriteEx()**: On input, the size of the `Buffer`. On output, the amount of data actually written. In both cases, the size is measured in bytes.

For **FlushEx()**: Not used, ignored

For **OpenEx()**: Not used, ignored

For **ReadEx()**: The buffer into which the data is read.

For **WriteEx()**: The buffer of data to write.

For **FlushEx()**: Not used, ignored

### Status Codes Returned

<table>
<thead>
<tr>
<th>Code</th>
<th>Description</th>
</tr>
</thead>
</table>
| EFI_SUCCESS              | Returned from the call **OpenEx()**  
If `Event` is NULL (blocking I/O): The file was opened successfully.  
If `Event` is not NULL (asynchronous I/O): The request was successfully queued for processing. `Event` will be signaled upon completion. Returned in the token after signaling `Event`  
The file was opened successfully. |
| EFI_NOT_FOUND            | The device has no medium.                                                   |
| EFI_NO_MEDIA             | The specified file could not be found on the device.                       |
| EFI_VOLUME_CORRUPTED     | The file system structures are corrupted.                                   |
| EFI_WRITE_PROTECTED      | An attempt was made to create a file, or open a file for write when the media is write-protected. |
| EFI_ACCESS_DENIED        | The service denied access to the file.                                      |
| EFI_OUT_OF_RESOURCES     | Unable to queue the request or open the file due to lack of resources.     |
| EFI_VOLUME_FULL          | The volume is full.                                                         |

### **EFI_FILE_PROTOCOL.ReadEx()**

**Summary**

Reads data from a file.

**Prototype**

```c
typedef EFI_STATUS (EFIAPI *EFI_FILE_READ_EX) (  
    IN EFI_FILE_PROTOCOL  *This,  
    IN OUT EFI_FILE_IO_TOKEN  *Token  
    );
```

**Parameters**

- **This**

A pointer to the `EFI_FILE_PROTOCOL` instance that is the file handle to read data from. See the type `EFI_FILE_PROTOCOL` description.
Token
A pointer to the token associated with the transaction. Type EFI_FILE_IO_TOKEN is defined in "Related Definitions" below.

Description
The ReadEx() function reads data from a file.

If This is not a directory, the function reads the requested number of bytes from the file at the file’s current position and returns them in Buffer. If the read goes beyond the end of the file, the read length is truncated to the end of the file. The file’s current position is increased by the number of bytes returned.

If This is a directory, the function reads the directory entry at the file’s current position and returns the entry in Buffer. If the Buffer is not large enough to hold the current directory entry, then EFI_BUFFER_TOO_SMALL is returned and the current file position is not updated. BufferSize is set to be the size of the buffer needed to read the entry. On success, the current position is updated to the next directory entry. If there are no more directory entries, the read returns a zero-length buffer.

EFI_FILE_INFO is the structure returned as the directory entry.

If non-blocking I/O is used the file pointer will be advanced based on the order that read requests were submitted.

If an error is returned from the call to ReadEx() and non-blocking I/O is being requested, the Event associated with this request will not be signaled. If the call to ReadEx() succeeds then the Event will be signaled upon completion of the read or if an error occurs during the processing of the request. The status of the read request can be determined from the Status field of the Token once the event is signaled.

Status Codes Returned

<table>
<thead>
<tr>
<th>Code</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>EFI_SUCCESS</td>
<td>Returned from the call ReadEx()</td>
</tr>
<tr>
<td></td>
<td>If Event is NULL (blocking I/O):</td>
</tr>
<tr>
<td></td>
<td>The data was read successfully.</td>
</tr>
<tr>
<td></td>
<td>If Event is not NULL (asynchronous I/O):</td>
</tr>
<tr>
<td></td>
<td>The request was successfully queued for processing. Event will be</td>
</tr>
<tr>
<td></td>
<td>signaled upon completion.</td>
</tr>
<tr>
<td></td>
<td>Returned in the token after signaling Event</td>
</tr>
<tr>
<td></td>
<td>The data was read successfully.</td>
</tr>
<tr>
<td>EFI_NO_MEDIA</td>
<td>The device has no medium.</td>
</tr>
<tr>
<td>EFI_DEVICE_ERROR</td>
<td>The device reported an error.</td>
</tr>
<tr>
<td>EFI_DEVICE_ERROR</td>
<td>An attempt was made to read from a deleted file.</td>
</tr>
<tr>
<td>EFI_VOLUME_CORRUPTED</td>
<td>On entry, the current file position is beyond the end of the file.</td>
</tr>
<tr>
<td>EFI_OUT_OF_RESOURCES</td>
<td>The file system structures are corrupted.</td>
</tr>
<tr>
<td></td>
<td>Unable to queue the request due to lack of resources.</td>
</tr>
</tbody>
</table>

EFI_FILE_PROTOCOL.WriteEx()

Summary
Writes data to a file.
Prototype

```c
typedef
  EFI_STATUS
  (EFIAPI *EFI_FILE_WRITE_EX) (  
    IN EFI_FILE_PROTOCOL    *This,
    IN OUT EFI_FILE_IO_TOKEN *Token
  );
```

Parameters

- **This**
  A pointer to the `EFI_FILE_PROTOCOL` instance that is the file handle to write data to. See the type `EFI_FILE_PROTOCOL` description.

- **Token**
  A pointer to the token associated with the transaction. Type `EFI_FILE_IO_TOKEN` is defined in "Related Definitions" above.

Description

The `WriteEx()` function writes the specified number of bytes to the file at the current file position. The current file position is advanced the actual number of bytes written, which is returned in `BufferSize`. Partial writes only occur when there has been a data error during the write attempt (such as “file space full”). The file is automatically grown to hold the data if required.

Direct writes to opened directories are not supported.

If non-blocking I/O is used the file pointer will be advanced based on the order that write requests were submitted.

If an error is returned from the call to `WriteEx()` and non-blocking I/O is being requested, the `Event` associated with this request will not be signaled. If the call to `WriteEx()` succeeds then the `Event` will be signaled upon completion of the write or if an error occurs during the processing of the request. The status of the write request can be determined from the `Status` field of the `Token` once the event is signaled.
**Status Codes Returned**

<table>
<thead>
<tr>
<th>Status Code</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>EFI_SUCCESS</td>
<td>Returned from the call WriteEx()</td>
</tr>
<tr>
<td></td>
<td>if Event is NULL (blocking I/O):</td>
</tr>
<tr>
<td></td>
<td>The data was written successfully.</td>
</tr>
<tr>
<td></td>
<td>if Event is not NULL (asynchronous I/O):</td>
</tr>
<tr>
<td></td>
<td>The request was successfully queued for processing. Event</td>
</tr>
<tr>
<td></td>
<td>will be signaled upon completion.</td>
</tr>
<tr>
<td></td>
<td>Returned in the token after signaling Event</td>
</tr>
<tr>
<td></td>
<td>The data was written successfully.</td>
</tr>
<tr>
<td>EFI_UNSUPPORTED</td>
<td>Writes to open directory files are not supported.</td>
</tr>
<tr>
<td>EFI_NO_MEDIA</td>
<td>The device has no medium.</td>
</tr>
<tr>
<td>EFI_DEVICE_ERROR</td>
<td>The device reported an error.</td>
</tr>
<tr>
<td>EFI_DEVICE_ERROR</td>
<td>An attempt was made to write to a deleted file.</td>
</tr>
<tr>
<td>EFI_VOLUME_CORRUPTED</td>
<td>The file system structures are corrupted.</td>
</tr>
<tr>
<td>EFI_WRITE_PROTECTED</td>
<td>The file or medium is write-protected.</td>
</tr>
<tr>
<td>EFI_ACCESS_DENIED</td>
<td>The file was opened read only.</td>
</tr>
<tr>
<td>EFI_VOLUME_FULL</td>
<td>The volume is full.</td>
</tr>
<tr>
<td>EFI_OUT_OF_RESOURCES</td>
<td>Unable to queue the request due to lack of resources.</td>
</tr>
</tbody>
</table>

**EFI_FILE_PROTOCOL.FlushEx()**

**Summary**
Flushes all modified data associated with a file to a device.

**Prototype**
```c
typedef
  EFI_STATUS
(EFIAPIC *EFI_FILE_FLUSH_EX) (  
    IN EFI_FILE_PROTOCOL  *This,
    IN OUT EFI_FILE_IO_TOKEN  *Token
  );
```

**Parameters**

- **This**
  A pointer to the `EFI_FILE_PROTOCOL` instance that is the file handle to flush. See the type `EFI_FILE_PROTOCOL` description.

- **Token**
  A pointer to the token associated with the transaction. Type `EFI_FILE_IO_TOKEN` is defined in "Related Definitions" above. The `BufferSize` and `Buffer` fields are not used for a FlushEx operation.

**Description**
The `FlushEx()` function flushes all modified data associated with a file to a device.

For non-blocking I/O all writes submitted before the flush request will be flushed.
If an error is returned from the call to `FlushEx()` and non-blocking I/O is being requested, the `Event` associated with this request will not be signaled.

### Status Codes Returned

<table>
<thead>
<tr>
<th>Status Code</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>EFI_SUCCESS</code></td>
<td>Returned from the call <code>FlushEx()</code></td>
</tr>
<tr>
<td></td>
<td>If <code>Event</code> is NULL (blocking I/O):</td>
</tr>
<tr>
<td></td>
<td>The data was flushed successfully.</td>
</tr>
<tr>
<td></td>
<td>If <code>Event</code> is not NULL (asynchronous I/O):</td>
</tr>
<tr>
<td></td>
<td>The request was successfully queued for processing. Event will</td>
</tr>
<tr>
<td></td>
<td>be signaled upon completion.</td>
</tr>
<tr>
<td></td>
<td>Returned in the token after signaling <code>Event</code></td>
</tr>
<tr>
<td></td>
<td>The data was flushed successfully.</td>
</tr>
<tr>
<td><code>EFI_NO_MEDIA</code></td>
<td>The device has no medium.</td>
</tr>
<tr>
<td><code>EFI_DEVICE_ERROR</code></td>
<td>The device reported an error.</td>
</tr>
<tr>
<td><code>EFI_VOLUME_CORRUPTED</code></td>
<td>The file system structures are corrupted.</td>
</tr>
<tr>
<td><code>EFI_WRITE_PROTECTED</code></td>
<td>The file or medium is write-protected.</td>
</tr>
<tr>
<td><code>EFI_ACCESS_DENIED</code></td>
<td>The file was opened read-only.</td>
</tr>
<tr>
<td><code>EFI_VOLUME_FULL</code></td>
<td>The volume is full.</td>
</tr>
<tr>
<td><code>EFI_OUT_OF_RESOURCES</code></td>
<td>Unable to queue the request due to lack of resources.</td>
</tr>
</tbody>
</table>

### `EFI_FILE_PROTOCOL.SetPosition()`

**Summary**

Sets a file’s current position.

**Prototype**

```c
typedef EFI_STATUS
(EFIAPI *EFI_FILE_SET_POSITION)(
    IN EFI_FILE_PROTOCOL *This,
    IN UINT64 Position
);
```

**Parameters**

- `This` A pointer to the `EFI_FILE_PROTOCOL` instance that is the file handle to set the requested position on. See the type `EFI_FILE_PROTOCOL` description.
- `Position` The byte position from the start of the file to set.

**Description**

The `SetPosition()` function sets the current file position for the handle to the position supplied. With the exception of seeking to position `0xFFFFFFFFFFFFFFFF`, only absolute positioning is supported, and seeking past the end of the file is allowed (a subsequent write would grow the file). Seeking to position `0xFFFFFFFFFFFFFFFF` causes the current position to be set to the end of the file.
If *This* is a directory, the only position that may be set is zero. This has the effect of starting the read process of the directory entries over.

**Status Codes Returned**

<table>
<thead>
<tr>
<th>Status Code</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>EFI_SUCCESS</td>
<td>The position was set.</td>
</tr>
<tr>
<td>EFI_UNSUPPORTED</td>
<td>The seek request for nonzero is not valid on open directories.</td>
</tr>
<tr>
<td>EFI_DEVICE_ERROR</td>
<td>An attempt was made to set the position of a deleted file.</td>
</tr>
</tbody>
</table>

**EFI_FILE_PROTOCOLGetPosition()**

**Summary**

Returns a file’s current position.

**Prototype**

```c
typedef EFI_STATUS (EFIAPI *EFI_FILE_GET_POSITION) (  
    IN EFI_FILE_PROTOCOL *This,  
    OUT UINT64 *Position  
);
```

**Parameters**

- *This* A pointer to the EFI_FILE_PROTOCOL instance that is the file handle to get the current position on. See the type EFI_FILE_PROTOCOL description.
- *Position* The address to return the file’s current position value.

**Description**

The `GetPosition()` function returns the current file position for the file handle. For directories, the current file position has no meaning outside of the file system driver and as such the operation is not supported. An error is returned if *This* is a directory.

**Status Codes Returned**

<table>
<thead>
<tr>
<th>Status Code</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>EFI_SUCCESS</td>
<td>The position was returned.</td>
</tr>
<tr>
<td>EFI_UNSUPPORTED</td>
<td>The request is not valid on open directories.</td>
</tr>
<tr>
<td>EFI_DEVICE_ERROR</td>
<td>An attempt was made to get the position from a deleted file.</td>
</tr>
</tbody>
</table>
EFI_FILE_PROTOCOL.GetInfo()

Summary
Returns information about a file.

Prototype
```
typedef
EFI_STATUS
(EIFIAPI *EFI_FILE_GET_INFO) (  
    IN EFI_FILE_PROTOCOL *This, 
    IN EFI_GUID *InformationType, 
    IN OUT UINTN *BufferSize, 
    OUT VOID *Buffer 
);
```

Parameters
- **This**: A pointer to the EFI_FILE_PROTOCOL instance that is the file handle the requested information is for. See the type EFI_FILE_PROTOCOL description.
- **InformationType**: The type identifier for the information being requested. Type EFI_GUID is defined on page 181. See the EFI_FILE_INFO and EFI_FILE_SYSTEM_INFO descriptions for the related GUID definitions.
- **BufferSize**: On input, the size of Buffer. On output, the amount of data returned in Buffer. In both cases, the size is measured in bytes.
- **Buffer**: A pointer to the data buffer to return. The buffer’s type is indicated by InformationType.

Description
The GetInfo() function returns information of type InformationType for the requested file. If the file does not support the requested information type, then EFI_UNSUPPORTED is returned. If the buffer is not large enough to fit the requested structure, EFI_BUFFER_TOO_SMALL is returned and the BufferSize is set to the size of buffer that is required to make the request.

The information types defined by this specification are required information types that all file systems must support.
Status Codes Returned

<table>
<thead>
<tr>
<th>Status Code</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>EFI_SUCCESS</td>
<td>The information was set.</td>
</tr>
<tr>
<td>EFI_UNSUPPORTED</td>
<td>The <code>InformationType</code> is not known.</td>
</tr>
<tr>
<td>EFI_NO_MEDIA</td>
<td>The device has no medium.</td>
</tr>
<tr>
<td>EFI_DEVICE_ERROR</td>
<td>The device reported an error.</td>
</tr>
<tr>
<td>EFI_VOLUME_CORRUPTED</td>
<td>The file system structures are corrupted.</td>
</tr>
<tr>
<td>EFI_BUFFER_TOO_SMALL</td>
<td>The <code>BufferSize</code> is too small to read the current directory entry. <code>BufferSize</code> has been updated with the size needed to complete the request.</td>
</tr>
</tbody>
</table>

**EFI_FILE_PROTOCOL.SetInfo()**

**Summary**
Sets information about a file.

**Prototype**

```c
typedef
EFI_STATUS
(EIFIAPI *EFI_FILE_SET_INFO) (  
    IN EFI_FILE_PROTOCOL *This,  
    IN EFI_GUID *InformationType,  
    IN UINTN BufferSize,  
    IN VOID *Buffer
);
```

**Parameters**

- **This**
  A pointer to the EFI_FILE_PROTOCOL instance that is the file handle the information is for. See the type EFI_FILE_PROTOCOL description.

- **InformationType**
  The type identifier for the information being set. Type EFI_GUID is defined in page 181. See the EFI_FILE_INFO and EFI_FILE_SYSTEM_INFO descriptions in this section for the related GUID definitions.

- **BufferSize**
  The size, in bytes, of `Buffer`.

- **Buffer**
  A pointer to the data buffer to write. The buffer’s type is indicated by `InformationType`.

**Description**
The `SetInfo()` function sets information of type `InformationType` on the requested file. Because a read-only file can be opened only in read-only mode, an `InformationType` of EFI_FILE_INFO_ID can be used with a read-only file because this method is the only one that can be used to convert a read-only file to a read-write file. In this circumstance, only the `Attribute` field of the EFI_FILE_INFO structure may be modified. One or more calls to `SetInfo()` to change the `Attribute` field are permitted before it is closed. The file attributes will be valid the next time the file is opened with `Open()`.
An **InformationType** of **EFI_FILE_SYSTEM_INFO_ID** or **EFI_FILE_SYSTEM_VOLUME_LABEL_ID** may not be used on read-only media.

**Status Codes Returned**

<table>
<thead>
<tr>
<th>Code</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>EFI_SUCCESS</td>
<td>The information was set.</td>
</tr>
<tr>
<td>EFI_UNSUPPORTED</td>
<td>The <strong>InformationType</strong> is not known.</td>
</tr>
<tr>
<td>EFI_NO_MEDIA</td>
<td>The device has no medium.</td>
</tr>
<tr>
<td>EFI_DEVICE_ERROR</td>
<td>The device reported an error.</td>
</tr>
<tr>
<td>EFI_VOLUME_CORRUPTED</td>
<td>The file system structures are corrupted.</td>
</tr>
<tr>
<td>EFI_WRITE_PROTECTED</td>
<td><strong>InformationType</strong> is <strong>EFI_FILE_INFO_ID</strong> and the media is read-only.</td>
</tr>
<tr>
<td>EFI_WRITE_PROTECTED</td>
<td><strong>InformationType</strong> is <strong>EFI_FILE_PROTOCOL_SYSTEM_INFO_ID</strong> and the media is read-only.</td>
</tr>
<tr>
<td>EFI_WRITE_PROTECTED</td>
<td><strong>InformationType</strong> is <strong>EFI_FILE_SYSTEM_VOLUME_LABEL_ID</strong> and the media is read-only.</td>
</tr>
<tr>
<td>EFI_ACCESS_DENIED</td>
<td>An attempt is made to change the name of a file to a file that is already present.</td>
</tr>
<tr>
<td>EFI_ACCESS_DENIED</td>
<td>An attempt is being made to change the <strong>EFI_FILE_DIRECTORY</strong> <strong>Attribute</strong>.</td>
</tr>
<tr>
<td>EFI_ACCESS_DENIED</td>
<td>An attempt is being made to change the size of a directory.</td>
</tr>
<tr>
<td>EFI_ACCESS_DENIED</td>
<td><strong>InformationType</strong> is <strong>EFI_FILE_INFO_ID</strong> and the file was opened read-only and an attempt is being made to modify a field other than <strong>Attribute</strong>.</td>
</tr>
<tr>
<td>EFI_VOLUME_FULL</td>
<td>The volume is full.</td>
</tr>
<tr>
<td>EFI_BAD_BUFFER_SIZE</td>
<td><strong>BufferSize</strong> is smaller than the size of the type indicated by <strong>InformationType</strong>.</td>
</tr>
</tbody>
</table>

**EFI_FILE_PROTOCOL.Flush()**

**Summary**

Flushes all modified data associated with a file to a device.

**Prototype**

```c
typedef
  EFI_STATUS
  (EFIAPI *EFI_FILE_FLUSH) (  
    IN EFI_FILE_PROTOCOL *This
  );
```

**Parameters**

- **This**
  
  A pointer to the **EFI_FILE_PROTOCOL** instance that is the file handle to flush. See the type **EFI_FILE_PROTOCOL** description.
Description

The `Flush()` function flushes all modified data associated with a file to a device.

Status Codes Returned

<table>
<thead>
<tr>
<th>Code</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>EFI_SUCCESS</td>
<td>The data was flushed.</td>
</tr>
<tr>
<td>EFI_NO_MEDIA</td>
<td>The device has no medium.</td>
</tr>
<tr>
<td>EFI_DEVICE_ERROR</td>
<td>The device reported an error.</td>
</tr>
<tr>
<td>EFI_VOLUME_CORRUPTED</td>
<td>The file system structures are corrupted.</td>
</tr>
<tr>
<td>EFI_WRITE_PROTECTED</td>
<td>The file or medium is write-protected.</td>
</tr>
<tr>
<td>EFI_ACCESS_DENIED</td>
<td>The file was opened read-only.</td>
</tr>
<tr>
<td>EFI_VOLUME_FULL</td>
<td>The volume is full.</td>
</tr>
</tbody>
</table>

`EFI_FILE_INFO`

Summary

Provides a GUID and a data structure that can be used with `EFI_FILE_PROTOCOL.SetInfo()` and `EFI_FILE_PROTOCOL.GetInfo()` to set or get generic file information.

GUID

```c
#define EFI_FILE_INFO_ID \  
{0x09576e92,0x6d3f,0x11d2,\  
{0x8e39,0x00,0xa0,0xc9,0x69,0x72,0x3b}}
```

Related Definitions

```c
typedef struct {
    UINT64 Size;
    UINT64 FileSize;
    UINT64 PhysicalSize;
    EFI_TIME CreateTime;
    EFI_TIME LastAccessTime;
    EFI_TIME ModificationTime;
    UINT64 Attribute;
    CHAR16 FileName[];
} EFI_FILE_INFO;
```

 EFI SUCCESS The data was flushed.
 EFI_NO_MEDIA The device has no medium.
 EFI_DEVICE_ERROR The device reported an error.
 EFI_VOLUME_CORRUPTED The file system structures are corrupted.
 EFI_WRITE_PROTECTED The file or medium is write-protected.
 EFI_ACCESS_DENIED The file was opened read-only.
 EFI_VOLUME_FULL The volume is full.

 EFI_WRITE_PROTECTED The file or medium is write-protected.
 EFI_ACCESS_DENIED The file was opened read-only.
 EFI_VOLUME_FULL The volume is full.
#define EFI_FILE_ARCHIVE     0x0000000000000020
#define EFI_FILE_VALID_ATTR  0x0000000000000037

Parameters

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Size</td>
<td>Size of the EFI_FILE_INFO structure, including the Null-terminated FileName string.</td>
</tr>
<tr>
<td>FileSize</td>
<td>The size of the file in bytes.</td>
</tr>
<tr>
<td>PhysicalSize</td>
<td>The amount of physical space the file consumes on the file system volume.</td>
</tr>
<tr>
<td>CreateTime</td>
<td>The time the file was created.</td>
</tr>
<tr>
<td>LastAccessTime</td>
<td>The time when the file was last accessed.</td>
</tr>
<tr>
<td>ModificationTime</td>
<td>The time when the file's contents were last modified.</td>
</tr>
<tr>
<td>Attribute</td>
<td>The attribute bits for the file. See “Related Definitions” above.</td>
</tr>
<tr>
<td>FileName</td>
<td>The Null-terminated name of the file. For a root directory, the name is an empty string.</td>
</tr>
</tbody>
</table>

Description

The EFI_FILE_INFO data structure supports GetInfo() and SetInfo() requests. In the case of SetInfo(), the following additional rules apply:

- On directories, the file size is determined by the contents of the directory and cannot be changed by setting FileSize. On directories, FileSize is ignored during a SetInfo().
- The PhysicalSize is determined by the FileSize and cannot be changed. This value is ignored during a SetInfo() request.
- The EFI_FILE_DIRECTORY attribute bit cannot be changed. It must match the file’s actual type.
- A value of zero in CreateTime, LastAccess, or ModificationTime causes the fields to be ignored (and not updated).

EFI_FILE_SYSTEM_INFO

Summary

Provides a GUID and a data structure that can be used with EFI_FILE_PROTOCOL.GetInfo() to get information about the system volume, and EFI_FILE_PROTOCOL.SetInfo() to set the system volume’s volume label.

GUID

```c
#define EFI_FILE_SYSTEM_INFO_ID \ 
{0x09576e93,0x6d3f,0x11d2,0x8e39,0x00,0xa0,0xc9,0x72,0x3b}
```
Related Definitions

typedef struct {
    UINT64   Size;
    BOOLEAN ReadOnly;
    UINT64   VolumeSize;
    UINT64   FreeSpace;
    UINT32   BlockSize;
    CHAR16   VolumeLabel[];
} EFI_FILE_SYSTEM_INFO;

Parameters

Size Size of the EFI_FILE_SYSTEM_INFO structure, including the Null-terminated VolumeLabel string.

ReadOnly TRUE if the volume only supports read access.

VolumeSize The number of bytes managed by the file system.

FreeSpace The number of available bytes for use by the file system.

BlockSize The nominal block size by which files are typically grown.

VolumeLabel The Null-terminated string that is the volume’s label.

Description

The EFI_FILE_SYSTEM_INFO data structure is an information structure that can be obtained on the root directory file handle. The root directory file handle is the file handle first obtained on the initial call to the EFI_BOOT_SERVICES.HandleProtocol() function to open the file system interface. All of the fields are read-only except for VolumeLabel. The system volume’s VolumeLabel can be created or modified by calling EFI_FILE_PROTOCOL.SetInfo() with an updated VolumeLabel field.

EFI_FILE_SYSTEM_VOLUME_LABEL

Summary

Provides a GUID and a data structure that can be used with EFI_FILE_PROTOCOL.GetInfo() or EFI_FILE_PROTOCOL.SetInfo() to get or set information about the system’s volume label.

GUID

#define EFI_FILE_SYSTEM_VOLUME_LABEL_ID \
{0xdb47d7d3,0xfe81,0x11d3,0x9a35,\n{0x00,0x90,0x27,0x3f,0xC1,0x4d}}
Related Definitions

typedef struct {
  CHAR16 VolumeLabel[];
} EFI_FILE_SYSTEM_VOLUME_LABEL;

Parameters

VolumeLabel The Null-terminated string that is the volume’s label.

Description

The EFI_FILE_SYSTEM_VOLUME_LABEL data structure is an information structure that can be obtained on the root directory file handle. The root directory file handle is the file handle first obtained on the initial call to the EFI_BOOT_SERVICES.HandleProtocol() function to open the file system interface. The system volume’s VolumeLabel can be created or modified by calling EFI_FILE_PROTOCOL.SetInfo() with an updated VolumeLabel field.

13.6 Tape Boot Support

13.6.1 Tape I/O Support

This section defines the Tape I/O Protocol and standard tape header format. These enable the support of booting from tape on UEFI systems. This protocol is used to abstract the tape drive operations to support applications written to this specification.

13.6.2 Tape I/O Protocol

This section defines the Tape I/O Protocol and its functions. This protocol is used to abstract the tape drive operations to support applications written to this specification.

EFI_TAPE_IO_PROTOCOL

Summary

The EFI Tape IO protocol provides services to control and access a tape device.

GUID

#define EFI_TAPE_IO_PROTOCOL_GUID \
  {0x1e93e633,0xd65a,0x459e, \
  {0xab,0x84,0x93,0xd9,0xec,0x26,0x18}}
Protocol Interface Structure

define struct _EFI_TAPE_IO_PROTOCOL {
    EFI_TAPE_READ        TapeRead;
    EFI_TAPE_WRITE       TapeWrite;
    EFI_TAPE_REWIND      TapeRewind;
    EFI_TAPE_SPACE       TapeSpace;
    EFI_TAPE_WRITEFM     TapeWriteFM;
    EFI_TAPE_RESET       TapeReset;
} EFI_TAPE_IO_PROTOCOL;

Parameters

TapeRead Read a block of data from the tape. See the TapeRead() description.

TapeWrite Write a block of data to the tape. See the TapeWrite() description.

TapeRewind Rewind the tape. See the TapeRewind() description.

TapeSpace Position the tape. See the TapeSpace() description.

TapeWriteFM Write filemarks to the tape. See the TapeWriteFM() description.

TapeReset Reset the tape device or its parent bus. See the TapeReset() description.

Description

The EFI_TAPE_IO_PROTOCOL provides basic sequential operations for tape devices. These include read, write, rewind, space, write filemarks and reset functions. Per this specification, a boot application uses the services of this protocol to load the bootloader image from tape.

No provision is made for controlling or determining media density or compression settings. The protocol relies on devices to behave normally and select settings appropriate for the media loaded. No support is included for tape partition support, setmarks or other tapemarks such as End of Data. Boot tapes are expected to use normal variable or fixed block size formatting and filemarks.
**EFI_TAPE_IO_PROTOCOL.TapeRead()**

**Summary**
Reads from the tape.

**Prototype**
```c
typedef EFI_STATUS (EFI_APFI *EFI_TAPE_READ) (
    IN EFI_TAPE_IO_PROTOCOL *This,
    IN OUT UINTN *BufferSize,
    OUT VOID *Buffer
);
```

**Parameters**
- **This**
  A pointer to the EFI_TAPE_IO_PROTOCOL instance.
- **BufferSize**
  Size of the buffer in bytes pointed to by Buffer.
- **Buffer**
  Pointer to the buffer for data to be read into.

**Description**
This function will read up to **BufferSize** bytes from media into the buffer pointed to by **Buffer** using an implementation-specific timeout. **BufferSize** will be updated with the number of bytes transferred.

Each read operation for a device that operates in variable block size mode reads one media data block. Unread bytes which do not fit in the buffer will be skipped by the next read operation. The number of bytes transferred will be limited by the actual media block size. Best practice is for the buffer size to match the media data block size. When a filemark is encountered in variable block size mode the read operation will indicate that 0 bytes were transferred and the function will return an EFI_END_OF_FILE error condition.

In fixed block mode the buffer is expected to be a multiple of the data block size. Each read operation for a device that operates in fixed block size mode may read multiple media data blocks. The number of bytes transferred will be limited to an integral number of complete media data blocks. **BufferSize** should be evenly divisible by the device’s fixed block size. When a filemark is encountered in fixed block size mode the read operation will indicate that the number of bytes transferred is less than the number of blocks that would fit in the provided buffer (possibly 0 bytes transferred) and the function will return an EFI_END_OF_FILE error condition.

Two consecutive filemarks are normally used to indicate the end of the last file on the media.

The value specified for **BufferSize** should correspond to the actual block size used on the media. If necessary, the value for **BufferSize** may be larger than the actual media block size.

Specifying a **BufferSize** of 0 is valid but requests the function to provide read-related status information instead of actual media data transfer. No data will be attempted to be read from the device however this operation is classified as an access for status handling. The status code returned may be used to determine if a filemark has been encountered by the last read request with a non-zero size, and to determine if media is loaded and the device is ready for reading. A NULL value for **Buffer** is valid when **BufferSize** is zero.
Status Codes Returned

<table>
<thead>
<tr>
<th>Status Code</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>EFI_SUCCESS</td>
<td>Data was successfully transferred from the media.</td>
</tr>
<tr>
<td>EFI_END_OF_FILE</td>
<td>A filemark was encountered which limited the data transferred by the read operation or the head is positioned just after a filemark.</td>
</tr>
<tr>
<td>EFI_NO_MEDIA</td>
<td>No media is loaded in the device.</td>
</tr>
<tr>
<td>EFI_MEDIA_CHANGED</td>
<td>The media in the device was changed since the last access. The transfer was aborted since the current position of the media may be incorrect.</td>
</tr>
<tr>
<td>EFI_DEVICE_ERROR</td>
<td>A device error occurred while attempting to transfer data from the media.</td>
</tr>
<tr>
<td>EFI_INVALID_PARAMETER</td>
<td>A NULL Buffer was specified with a non-zero BufferSize or the device is operating in fixed block size mode and the BufferSize was not a multiple of device’s fixed block size.</td>
</tr>
<tr>
<td>EFI_NOT_READY</td>
<td>The transfer failed since the device was not ready (e.g. not online). The transfer may be retried at a later time.</td>
</tr>
<tr>
<td>EFI_UNSUPPORTED</td>
<td>The device does not support this type of transfer.</td>
</tr>
<tr>
<td>EFI_TIMEOUT</td>
<td>The transfer failed to complete within the timeout specified.</td>
</tr>
</tbody>
</table>

EFI_TAPE_IO_PROTOCOL.TapeWrite()

Summary
Write to the tape.

Prototype

```c
typedef EFI_STATUS
    (EFIAPI *EFI_TAPE_WRITE) (
    IN EFI_TAPE_IO_PROTOCOL *This,
    IN UINTN BufferSize,
    IN VOID *Buffer
    );
```

Parameters

- This: A pointer to the EFI_TAPE_IO_PROTOCOL instance.
- BufferSize: Size of the buffer in bytes pointed to by Buffer.
- Buffer: Pointer to the buffer for data to be written from.

Description

This function will write BufferSize bytes from the buffer pointed to by Buffer to media using an implementation-specific timeout.

Each write operation for a device that operates in variable block size mode writes one media data block of BufferSize bytes.

Each write operation for a device that operates in fixed block size mode writes one or more media data blocks of the device’s fixed block size. BufferSize must be evenly divisible by the device’s fixed block size.
Although sequential devices in variable block size mode support a wide variety of block sizes, many issues may be avoided in I/O software, adapters, hardware and firmware if common block sizes are used such as: 32768, 16384, 8192, 4096, 2048, 1024, 512, and 80.

**BufferSize** will be updated with the number of bytes transferred.

When a write operation occurs beyond the logical end of media an **EFI_END_OF_MEDIA** error condition will occur. Normally data will be successfully written and **BufferSize** will be updated with the number of bytes transferred. Additional write operations will continue to fail in the same manner. Excessive writing beyond the logical end of media should be avoided since the physical end of media may be reached.

Specifying a **BufferSize** of 0 is valid but requests the function to provide write-related status information instead of actual media data transfer. No data will be attempted to be written to the device however this operation is classified as an access for status handling. The status code returned may be used to determine if media is loaded, writable and if the logical end of media point has been reached. A **NULL** value for **Buffer** is valid when **BufferSize** is zero.

### Status Codes Returned

<table>
<thead>
<tr>
<th>Status Code</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>EFI_SUCCESS</td>
<td>Data was successfully transferred to the media.</td>
</tr>
<tr>
<td>EFI_END_OF_MEDIA</td>
<td>The logical end of media has been reached. Data may have been successfully transferred to the media.</td>
</tr>
<tr>
<td>EFI_NO_MEDIA</td>
<td>No media is loaded in the device.</td>
</tr>
<tr>
<td>EFI_MEDIA_CHANGED</td>
<td>The media in the device was changed since the last access. The transfer was aborted since the current position of the media may be incorrect.</td>
</tr>
<tr>
<td>EFI_WRITE_PROTECTED</td>
<td>The media in the device is write-protected. The transfer was aborted since a write cannot be completed.</td>
</tr>
<tr>
<td>EFI_DEVICE_ERROR</td>
<td>A device error occurred while attempting to transfer data from the media.</td>
</tr>
<tr>
<td>EFI_INVALID_PARAMETER</td>
<td>A <strong>NULL Buffer</strong> was specified with a non-zero <strong>BufferSize</strong> or the device is operating in fixed block size mode and <strong>BufferSize</strong> was not a multiple of device’s fixed block size.</td>
</tr>
<tr>
<td>EFI_NOT_READY</td>
<td>The transfer failed since the device was not ready (e.g. not online). The transfer may be retried at a later time.</td>
</tr>
<tr>
<td>EFI_UNSUPPORTED</td>
<td>The device does not support this type of transfer.</td>
</tr>
<tr>
<td>EFI_TIMEOUT</td>
<td>The transfer failed to complete within the timeout specified.</td>
</tr>
</tbody>
</table>

### EFI_TAPE_IO_PROTOCOL.TapeRewind()

**Summary**

Rewinds the tape.
Prototype

typedef
EFI_STATUS
(EIFI_API *EFI_TAPE_REWIND) (  
    IN EFI_TAPE_IO_PROTOCOL  *This,
);

Parameters

This          A pointer to the EFI_TAPE_IO_PROTOCOL instance.

Description

This function will rewind the media using an implementation-specific timeout. The function will check if
the media was changed since the last access and reinstall the EFI_TAPE_IO_PROTOCOL interface for the
device handle if needed.

Status Codes Returned

<table>
<thead>
<tr>
<th>Code</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>EFI_SUCCESS</td>
<td>The media was successfully repositioned.</td>
</tr>
<tr>
<td>EFI_NO_MEDIA</td>
<td>No media is loaded in the device.</td>
</tr>
<tr>
<td>EFI_DEVICE_ERROR</td>
<td>A device error occurred while attempting to reposition the media.</td>
</tr>
<tr>
<td>EFI_NOT_READY</td>
<td>Repositioning the media failed since the device was not ready (e.g. not</td>
</tr>
<tr>
<td></td>
<td>online). The transfer may be retried at a later time.</td>
</tr>
<tr>
<td>EFI_UNSUPPORTED</td>
<td>The device does not support this type of media repositioning.</td>
</tr>
<tr>
<td>EFI_TIMEOUT</td>
<td>Repositioning of the media did not complete within the timeout specified.</td>
</tr>
</tbody>
</table>

EFI_TAPE_IO_PROTOCOL.TapeSpace()

Summary

Positions the tape.

Prototype

typedef
EFI_STATUS
(EIFI_API *EFI_TAPE_SPACE) (  
    IN EFI_TAPE_IO_PROTOCOL  *This,
    IN INTN               Direction,
    IN UINTN              Type
);

Parameters

This          A pointer to the EFI_TAPE_IO_PROTOCOL instance.

Direction     Direction and number of data blocks or filemarks to space over on
media.

Type          Type of mark to space over on media.
Description

This function will position the media using an implementation-specific timeout. A positive Direction value will indicate the number of data blocks or filemarks to forward space the media. A negative Direction value will indicate the number of data blocks or filemarks to reverse space the media.

The following Type marks are mandatory:

<table>
<thead>
<tr>
<th>Type of Tape Mark</th>
<th>MarkType</th>
</tr>
</thead>
<tbody>
<tr>
<td>BLOCK</td>
<td>0</td>
</tr>
<tr>
<td>FILEMARK</td>
<td>1</td>
</tr>
</tbody>
</table>

Space operations position the media past the data block or filemark. Forward space operations leave media positioned with the tape device head after the data block or filemark. Reverse space operations leave the media positioned with the tape device head before the data block or filemark.

If beginning of media is reached before a reverse space operation passes the requested number of data blocks or filemarks an EFI_END_OF_MEDIA error condition will occur. If end of recorded data or end of physical media is reached before a forward space operation passes the requested number of data blocks or filemarks an EFI_END_OF_MEDIA error condition will occur. An EFI_END_OF_MEDIA error condition will not occur due to spacing over data blocks or filemarks past the logical end of media point used to indicate when write operations should be limited.

Status Codes Returned

<table>
<thead>
<tr>
<th>Status Code</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>EFI_SUCCESS</td>
<td>The media was successfully repositioned.</td>
</tr>
<tr>
<td>EFI_END_OF_MEDIA</td>
<td>Beginning or end of media was reached before the indicated number of data blocks or filemarks were found.</td>
</tr>
<tr>
<td>EFI_NO_MEDIA</td>
<td>No media is loaded in the device.</td>
</tr>
<tr>
<td>EFI_MEDIA_CHANGED</td>
<td>The media in the device was changed since the last access. Repositioning the media was aborted since the current position of the media may be incorrect.</td>
</tr>
<tr>
<td>EFI_DEVICE_ERROR</td>
<td>A device error occurred while attempting to reposition the media.</td>
</tr>
<tr>
<td>EFI_NOT_READY</td>
<td>Repositioning the media failed since the device was not ready (e.g. not online). The transfer may be retried at a later time.</td>
</tr>
<tr>
<td>EFI_UNSUPPORTED</td>
<td>The device does not support this type of media repositioning.</td>
</tr>
<tr>
<td>EFI_TIMEOUT</td>
<td>Repositioning of the media did not complete within the timeout specified.</td>
</tr>
</tbody>
</table>

EFI_TAPE_IO_PROTOCOL.TapeWriteFM()

Summary

Writes filemarks to the media.
Prototype

typedef
EFI_STATUS
(EIFIAPI *EFI_TAPE_WRITEFM) (
    IN EFI_TAPE_IO_PROTOCOL *This,
    IN UINTN Count
);

Parameters
This A pointer to the EFI_TAPE_IO_PROTOCOL instance.
Count Number of filemarks to write to the media.

Description
This function will write filemarks to the tape using an implementation-specific timeout.

Writing filemarks beyond logical end of tape does not result in an error condition unless physical end of media is reached.

Status Codes Returned

<table>
<thead>
<tr>
<th>Code</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>EFI_SUCCESS</td>
<td>Data was successfully transferred from the media.</td>
</tr>
<tr>
<td>EFI_NO_MEDIA</td>
<td>No media is loaded in the device.</td>
</tr>
<tr>
<td>EFI_MEDIA_CHANGED</td>
<td>The media in the device was changed since the last access. The transfer was aborted since the current position of the media may be incorrect.</td>
</tr>
<tr>
<td>EFI_DEVICE_ERROR</td>
<td>A device error occurred while attempting to transfer data from the media.</td>
</tr>
<tr>
<td>EFI_NOT_READY</td>
<td>The transfer failed since the device was not ready (e.g. not online). The transfer may be retried at a later time.</td>
</tr>
<tr>
<td>EFI_UNSUPPORTED</td>
<td>The device does not support this type of transfer.</td>
</tr>
<tr>
<td>EFI_TIMEOUT</td>
<td>The transfer failed to complete within the timeout specified.</td>
</tr>
</tbody>
</table>

EFI_TAPE_IO_PROTOCOL.TapeReset()

Summary
Resets the tape device.

Prototype

typedef
EFI_STATUS
(EIFIAPI *EFI_TAPE_RESET) (
    IN EFI_TAPE_IO_PROTOCOL *This,
    IN BOOLEAN ExtendedVerification
);

Parameters
This A pointer to the EFI_TAPE_IO_PROTOCOL instance.
**ExtendedVerification**
Indicates whether the parent bus should also be reset.

**Description**
This function will reset the tape device. If *ExtendedVerification* is set to true, the function will reset the parent bus (e.g., SCSI bus). The function will check if the media was changed since the last access and reinstall the EFI_TAPE_IO_PROTOCOL interface for the device handle if needed. Note media needs to be loaded and device online for the reset, otherwise, EFI_DEVICE_ERROR is returned.

**Status Codes Returned**

<table>
<thead>
<tr>
<th>Code</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>EFI_SUCCESS</td>
<td>The bus and/or device were successfully reset.</td>
</tr>
<tr>
<td>EFI_NO_MEDIA</td>
<td>No media is loaded in the device.</td>
</tr>
<tr>
<td>EFI_DEVICE_ERROR</td>
<td>A device error occurred while attempting to reset the bus and/or device.</td>
</tr>
<tr>
<td>EFI_NOT_READY</td>
<td>The reset failed since the device and/or bus was not ready. The reset may be retried at a later time.</td>
</tr>
<tr>
<td>EFI_UNSUPPORTED</td>
<td>The device does not support this type of reset.</td>
</tr>
<tr>
<td>EFI_TIMEOUT</td>
<td>The reset did not complete within the timeout allowed.</td>
</tr>
</tbody>
</table>

### 13.6.3 Tape Header Format

The boot tape will contain a Boot Tape Header to indicate it is a valid boot tape. The Boot Tape Header must be located within the first 20 blocks on the tape. One or more tape filemarks may appear prior to the Boot Tape Header so that boot tapes may include tape label files. The Boot Tape Header must begin on a block boundary and be contained completely within a block. The Boot Tape Header will have the following format:

#### Table 13-1 Tape Header Formats

<table>
<thead>
<tr>
<th>Bytes (Dec)</th>
<th>Value</th>
<th>Purpose</th>
</tr>
</thead>
<tbody>
<tr>
<td>0-7</td>
<td>0x544f4f4220494645</td>
<td>Signature (‘EFI BOOT’ in ASCII)</td>
</tr>
<tr>
<td>8-11</td>
<td>1</td>
<td>Revision</td>
</tr>
<tr>
<td>12-15</td>
<td>1024</td>
<td>Tape Header Size in bytes</td>
</tr>
<tr>
<td>16-19</td>
<td>calculated</td>
<td>Tape Header CRC</td>
</tr>
<tr>
<td>20-35</td>
<td>{ 0x8befa29a, 0x3511, 0x4cf7, { 0xa2, 0xeb, 0x5f, 0xe3, 0x7c, 0x3b, 0xf5, 0x5b } }</td>
<td>EFI Boot Tape GUID (same for all EFI Boot Tapes, like EFI Disk GUID)</td>
</tr>
<tr>
<td>36-51</td>
<td>User Defined</td>
<td>EFI Boot Tape Type GUID (bootloader / OS specific, like EFI Partition Type GUID)</td>
</tr>
<tr>
<td>52-67</td>
<td>User Defined</td>
<td>EFI Boot Tape Unique GUID (unique for every EFI Boot Tape)</td>
</tr>
<tr>
<td>68-71</td>
<td>e.g. 2</td>
<td>File Number of EFI Bootloader relative to the Boot Tape Header (first file immediately after the Boot Tape Header is file number 1, ANSI labels are counted)</td>
</tr>
<tr>
<td>72-75</td>
<td>e.g. 0x400</td>
<td>EFI Bootloader Block Size in bytes</td>
</tr>
<tr>
<td>76-79</td>
<td>e.g. 0x20000</td>
<td>EFI Bootloader Total Size in bytes</td>
</tr>
</tbody>
</table>
All numeric values will be specified in binary format. Note that all values are specified in Little Endian byte ordering.

The Boot Tape Header can also be represented as the following data structure:

```c
typedef struct EFI_TAPE_HEADER {
    UINT64    Signature;
    UINT32    Revision;
    UINT32    BootDescSize;
    UINT32    BootDescCRC;
    EFI_GUID  TapeGUID;
    EFI_GUID  TapeType;
    EFI_GUID  TapeUnique;
    UINT32    BLLocation;
    UINT32    BLBlocksize;
    UINT32    BLFilesize;
    CHAR8     OSVersion[40];
    CHAR8     AppVersion[40];
    CHAR8     CreationDate[10];
    CHAR8     CreationTime[10];
    CHAR8     SystemName[256];  // UTF-8
    CHAR8     TapeTitle[120];   // UTF-8
    CHAR8     pad[468];   // pad to 1024
} EFI_TAPE_HEADER;
```

### 13.7 Disk I/O Protocol

This section defines the Disk I/O protocol. This protocol is used to abstract the block accesses of the Block I/O protocol to a more general offset-length protocol. The firmware is responsible for adding this protocol to any Block I/O interface that appears in the system that does not already have a Disk I/O protocol. File systems and other disk access code utilize the Disk I/O protocol.
**EFI_DISK_IO_PROTOCOL**

**Summary**
This protocol is used to abstract Block I/O interfaces.

**GUID**
```c
#define EFI_DISK_IO_PROTOCOL_GUID \
{0xCE345171,0xBA0B,0x11d2,\ 
 {0x8e,0x4F,0x00,0xa0,0xc9,0x72,0x3b}}
```

**Revision Number**
```c
#define EFI_DISK_IO_PROTOCOL_REVISION 0x00010000
```

**Protocol Interface Structure**
```c
typedef struct _EFI_DISK_IO_PROTOCOL {
    UINT64 Revision;
    EFI_DISK_READ ReadDisk;
    EFI_DISK_WRITE WriteDisk;
} EFI_DISK_IO_PROTOCOL;
```

**Parameters**
- **Revision**
  The revision to which the disk I/O interface adheres. All future revisions must be backwards compatible. If a future version is not backwards compatible, it is not the same GUID.
- **ReadDisk**
  Reads data from the disk. See the `ReadDisk()` function description.
- **WriteDisk**
  Writes data to the disk. See the `WriteDisk()` function description.

**Description**
The **EFI_DISK_IO_PROTOCOL** is used to control block I/O interfaces.

The disk I/O functions allow I/O operations that need not be on the underlying device’s block boundaries or alignment requirements. This is done by copying the data to/from internal buffers as needed to provide the proper requests to the block I/O device. Outstanding write buffer data is flushed by using the `FlushBlocks()` function of the **EFI_BLOCK_IO_PROTOCOL** on the device handle.

The firmware automatically adds a **EFI_DISK_IO_PROTOCOL** interface to any **EFI_BLOCK_IO_PROTOCOL** interface that is produced. It also adds file system, or logical block I/O, interfaces to any **EFI_DISK_IO_PROTOCOL** interface that contains any recognized file system or logical block I/O devices. The firmware must automatically support the following required formats:

- The EFI FAT12, FAT16, and FAT32 file system type.
- The legacy master boot record partition block. (The presence of this on any block I/O device is optional, but if it is present the firmware is responsible for allocating a logical device for each partition).
- The extended partition record partition block.
- The El Torito logical block devices.
EFI_DISK_IO_PROTOCOL.ReadDisk()

Summary
Reads a specified number of bytes from a device.

Prototype

```c
typedef EFI_STATUS (EFIAPI *EFI_DISK_READ) (  
    IN EFI_DISK_IO_PROTOCOL *This,
    IN UINT32 MediaId,
    IN UINT64 Offset,
    IN UINTN BufferSize,
    OUT VOID *Buffer
);
```

Parameters

- **This**: Indicates a pointer to the calling context. Type `EFI_DISK_IO_PROTOCOL` is defined in the `EFI_DISK_IO_PROTOCOL` description.
- **MediaId**: ID of the medium to be read.
- **Offset**: The starting byte offset on the logical block I/O device to read from.
- **BufferSize**: The size in bytes of `Buffer`. The number of bytes to read from the device.
- **Buffer**: A pointer to the destination buffer for the data. The caller is responsible for either having implicit or explicit ownership of the buffer.

Description

The `ReadDisk()` function reads the number of bytes specified by `BufferSize` from the device. All the bytes are read, or an error is returned. If there is no medium in the device, the function returns `EFI_NO_MEDIA`. If the `MediaId` is not the ID of the medium currently in the device, the function returns `EFI_MEDIA_CHANGED`. 
## Status Codes Returned

<table>
<thead>
<tr>
<th>Code</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>EFI_SUCCESS</td>
<td>The data was read correctly from the device.</td>
</tr>
<tr>
<td>EFI DEVICE_ERROR</td>
<td>The device reported an error while performing the read operation.</td>
</tr>
<tr>
<td>EFI_NO_MEDIA</td>
<td>There is no medium in the device.</td>
</tr>
<tr>
<td>EFI MEDIA_CHANGED</td>
<td>The MediaId is not for the current medium.</td>
</tr>
<tr>
<td>EFI INVALID_PARAMETER</td>
<td>The read request contains device addresses that are not valid for the device.</td>
</tr>
</tbody>
</table>

### EFI_DISK_IO_PROTOCOL.WriteDisk()

#### Summary

Writes a specified number of bytes to a device.

#### Prototype

```c
typedef EFI_STATUS
    (EFIAPI *EFI_DISK_WRITE) (    
    IN EFI_DISK_IO_PROTOCOL *This,    
    IN UINT32 MediaId,    
    IN UINT64 Offset,    
    IN UINTN BufferSize,    
    IN VOID *Buffer    
    );
```

#### Parameters

- **This**
  - Indicates a pointer to the calling context. Type `EFI_DISK_IO_PROTOCOL` is defined in the `EFI_DISK_IO_PROTOCOL` protocol description.
- **MediaId**
  - ID of the medium to be written.
- **Offset**
  - The starting byte offset on the logical block I/O device to write.
- **BufferSize**
  - The size in bytes of `Buffer`. The number of bytes to write to the device.
- **Buffer**
  - A pointer to the buffer containing the data to be written.

#### Description

The `WriteDisk()` function writes the number of bytes specified by `BufferSize` to the device. All bytes are written, or an error is returned. If there is no medium in the device, the function returns `EFI_NO_MEDIA`. If the `MediaId` is not the ID of the medium currently in the device, the function returns `EFI_MEDIA_CHANGED`. 
### Status Codes Returned

<table>
<thead>
<tr>
<th>Code</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>EFI_SUCCESS</td>
<td>The data was written correctly to the device.</td>
</tr>
<tr>
<td>EFI_WRITE_PROTECTED</td>
<td>The device cannot be written to.</td>
</tr>
<tr>
<td>EFI_NO_MEDIA</td>
<td>There is no medium in the device.</td>
</tr>
<tr>
<td>EFI_MEDIA_CHANGED</td>
<td>The MediaId is not for the current medium.</td>
</tr>
<tr>
<td>EFI_DEVICE_ERROR</td>
<td>The device reported an error while performing the write operation.</td>
</tr>
<tr>
<td>EFI_INVALID_PARAMETER</td>
<td>The write request contains device addresses that are not valid for the device.</td>
</tr>
</tbody>
</table>

### 13.8 Disk I/O 2 Protocol

The Disk I/O 2 protocol defines an extension to the Disk I/O protocol to enable non-blocking / asynchronous byte-oriented disk operation.

#### EFI_DISK_IO2_PROTOCOL

**Summary**

This protocol is used to abstract Block I/O interfaces in a non-blocking manner.

**GUID**

```c
#define EFI_DISK_IO2_PROTOCOL_GUID  
{ 0x151c8eae, 0x7f2c, 0x472c,  
{0x9e, 0x54, 0x98, 0x28, 0x19, 0x4f, 0x6a, 0x88 }}
```

**Revision Number**

```c
#define EFI_DISK_IO2_PROTOCOL_REVISION 0x00020000
```

**Protocol Interface Structure**

```c
typedef struct _EFI_DISK_IO2_PROTOCOL {  
    UINT64 Revision;  
    EFI_DISK_CANCEL_EX Cancel;  
    EFI_DISK_READ_EX ReadDiskEx;  
    EFI_DISK_WRITE_EX WriteDiskEx;  
    EFI_DISK_FLUSH_EX FlushDiskEx;  
} EFI_DISK_IO2_PROTOCOL;
```

**Parameters**

- **Revision**
  
  The revision to which the disk I/O interface adheres. All future revisions must be backwards compatible.

- **Cancel**
  
  Terminate outstanding requests. See the `Cancel()` function description.

- **ReadDiskEx**
  
  Reads data from the disk. See the `ReadDiskEx()` function description.
WriteDiskEx  Writes data to the disk. See the WriteDiskEx() function description.

FlushDiskEx  Flushes all modified data to the physical device. See the FlushDiskEx() function description.

Description
The EFI_DISK_IO2_PROTOCOL is used to control block I/O interfaces.

The disk I/O functions allow I/O operations that need not be on the underlying device’s block boundaries or alignment requirements. This is done by copying the data to/from internal buffers as needed to provide the proper requests to the block I/O device. Outstanding write buffer data is flushed by using the FlushBlocksEx() function of the EFI_BLOCK_IO2_PROTOCOL on the device handle.

The firmware automatically adds an EFI_DISK_IO2_PROTOCOL interface to any EFI_BLOCK_IO2_PROTOCOL interface that is produced. It also adds file system, or logical block I/O, interfaces to any EFI_DISK_IO2_PROTOCOL interface that contains any recognized file system or logical block I/O devices.

Implementations must account for cases where there is pending queued asynchronous I/O when a call is received on a blocking protocol interface. In these cases the pending I/O will be processed and completed before the blocking function is executed so that operation are carried out in the order they were requested.

EFI_DISK_IO2_PROTOCOL.Cancel()

Summary
Terminate outstanding asynchronous requests to a device.

Prototype
typedef EFI_STATUS
(EFIAPIC *EFI_DISK_CANCEL_EX) (
    IN EFI_DISK_IO2_PROTOCOL *This
);

Parameters
This  Indicates a pointer to the calling context. Type EFI_DISK_IO2_PROTOCOL is defined in the EFI_DISK_IO2_PROTOCOL description.

Description
The Cancel() function will terminate any in-flight non-blocking I/O requests by signaling the EFI_DISK_IO2_TOKEN Event and with TransactionStatus set to EFI_ABORTED. After the Cancel() function returns it is safe to free any Token or Buffer data structures that were allocated as part of the non-blocking I/O operation.
Status Codes Returned

<table>
<thead>
<tr>
<th>Status Code</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>EFI_SUCCESS</td>
<td>All outstanding requests were successfully terminated.</td>
</tr>
<tr>
<td>EFI_DEVICE_ERROR</td>
<td>The device reported an error while performing the cancel operation.</td>
</tr>
</tbody>
</table>

**EFI_DISK_IO2_PROTOCOL.ReadDiskEx()**

**Summary**
Reads a specified number of bytes from a device.

**Prototype**
```c
typedef EFI_STATUS
    (EFIAPI *EFI_DISK_READ_EX) (    
        IN EFI_DISK_IO2_PROTOCOL    *This,    
        IN UINT32                   MediaId,    
        IN UINT64                   Offset,    
        IN OUT EFI_DISK_IO2_TOKEN   *Token,    
        IN UINTN                    BufferSize,    
        OUT VOID                    *Buffer    
    );
```

**Parameters**
- **This**
  Indicates a pointer to the calling context. Type `EFI_DISK_IO2_PROTOCOL` is defined in the `EFI_DISK_IO2_PROTOCOL` description.
- **MediaId**
  ID of the medium to be read.
- **Offset**
  The starting byte offset on the logical block I/O device to read from.
- **Token**
  A pointer to the token associated with the transaction. Type `EFI_DISK_IO2_TOKEN` is defined in "Related Definitions" below. If this field is NULL, synchronous/blocking IO is performed.
- **BufferSize**
  The size in bytes of `Buffer`. The number of bytes to read from the device.
- **Buffer**
  A pointer to the destination buffer for the data. The caller is responsible either having implicit or explicit ownership of the buffer.

**Description**
The `ReadDiskEx()` function reads the number of bytes specified by `BufferSize` from the device. All the bytes are read, or an error is returned. If there is no medium in the device, the function returns `EFI_NO_MEDIA`. If the `MediaId` is not the ID of the medium currently in the device, the function returns `EFI_MEDIA_CHANGED`.

If an error is returned from the call to `ReadDiskEx()` and non-blocking I/O is being requested, the `Event` associated with this request will not be signaled. If the call to `ReadDiskEx()` succeeds then the `Event` will be signaled upon completion of the read or if an error occurs during the processing of the
request. The status of the read request can be determined from the `Status` field of the `Token` once the event is signaled.

**Related Definitions**

```c
typedef struct {
    EFI_EVENT Event;
    EFI_STATUS TransactionStatus;
} EFI_DISK_IO2_TOKEN;
```

*Event* If `Event` is NULL, then blocking I/O is performed. If `Event` is not NULL and non-blocking I/O is supported, then non-blocking I/O is performed, and `Event` will be signaled when the I/O request is completed. The caller must be prepared to handle the case where the callback associated with `Event` occurs before the original asynchronous I/O request call returns.

*TransactionStatus* Defines whether or not the signaled event encountered an error.

**Status Codes Returned**

<table>
<thead>
<tr>
<th>Code</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>EFI_SUCCESS</td>
<td>Returned from the call <code>ReadDiskEx()</code>&lt;br&gt; If <code>Event</code> is NULL (blocking I/O):&lt;br&gt;The data was read correctly from the device.&lt;br&gt; If <code>Event</code> is not NULL (asynchronous I/O):&lt;br&gt;The request was successfully queued for processing. <code>Event</code> will be signaled upon completion.&lt;br&gt;Returned in the token after signaling <code>Event</code>. The data was read correctly from the device.</td>
</tr>
<tr>
<td>EFI_DEVICE_ERROR</td>
<td>The device reported an error while performing the read operation.</td>
</tr>
<tr>
<td>EFI_NO_MEDIA</td>
<td>There is no medium in the device.</td>
</tr>
<tr>
<td>EFI_MEDIA_CHANGED</td>
<td>The <code>MediaId</code> is not for the current medium.</td>
</tr>
<tr>
<td>EFI_INVALID_PARAMETER</td>
<td>The read request contains device addresses that are not valid for the device.</td>
</tr>
<tr>
<td>EFI_OUT_OF_RESOURCES</td>
<td>The request could not be completed due to a lack of resources.</td>
</tr>
</tbody>
</table>

**EFI_DISK_IO2_PROTOCOL.WriteDiskEx()**

**Summary**

Writes a specified number of bytes to a device.
Prototype

```c
typedef EFI_STATUS (EFIAPI *EFI_DISK_WRITE_EX) (
    IN EFI_DISK_IO2_PROTOCOL *This,
    IN UINT32 MediaId,
    IN UINT64 Offset,
    IN OUT EFI_DISK_IO2_TOKEN *Token,
    IN UINTN BufferSize,
    IN VOID *Buffer
);
```

Parameters

- **This**: Indicates a pointer to the calling context. Type `EFI_DISK_IO2_PROTOCOL` is defined in the `EFI_DISK_IO2_PROTOCOL` description.
- **MediaId**: ID of the medium to be written.
- **Offset**: The starting byte offset on the logical block I/O device to write to.
- **Token**: A pointer to the token associated with the transaction. Type `EFI_DISK_IO2_TOKEN` is defined in "Related Definitions" below. If this field is NULL, synchronous/blocking IO is performed.
- **BufferSize**: The size in bytes of `Buffer`. The number of bytes to write to the device.
- **Buffer**: A pointer to the source buffer for the data. The caller is responsible.

Description

The `WriteDiskEx()` function writes the number of bytes specified by `BufferSize` to the device. All bytes are written, or an error is returned. If there is no medium in the device, the function returns `EFI_NO_MEDIA`. If the `MediaId` is not the ID of the medium currently in the device, the function returns `EFI_MEDIA_CHANGED`.

If an error is returned from the call to `WriteDiskEx()` and non-blocking I/O is being requested, the `Event` associated with this request will not be signaled. If the call to `WriteDiskEx()` succeeds then the `Event` will be signaled upon completion of the write or if an error occurs during the processing of the request. The status of the write request can be determined from the `Status` field of the `Token` once the event is signaled.
Status Codes Returned

<table>
<thead>
<tr>
<th>Code</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>EFI_SUCCESS</td>
<td>Returned from the call <strong>WriteDiskEx()</strong></td>
</tr>
<tr>
<td></td>
<td>If <strong>Event</strong> is NULL (blocking I/O):</td>
</tr>
<tr>
<td></td>
<td>• The data was written correctly to the device.</td>
</tr>
<tr>
<td></td>
<td>If <strong>Event</strong> is not NULL (asynchronous I/O):</td>
</tr>
<tr>
<td></td>
<td>• The request was successfully queued for processing. <strong>Event</strong> will be</td>
</tr>
<tr>
<td></td>
<td>signaled upon completion.</td>
</tr>
<tr>
<td>EFI_WRITE_PROTECTED</td>
<td>The device cannot be written to.</td>
</tr>
<tr>
<td>EFI_DEVICE_ERROR</td>
<td>The device reported an error while performing the write operation.</td>
</tr>
<tr>
<td>EFI_NO_MEDIA</td>
<td>There is no medium in the device.</td>
</tr>
<tr>
<td>EFI_MEDIA_CHANGED</td>
<td>The <strong>MediaId</strong> is not for the current medium.</td>
</tr>
<tr>
<td>EFI_INVALID_PARAMETER</td>
<td>The read request contains device addresses that are not valid for the device.</td>
</tr>
<tr>
<td>EFI_OUT_OF_RESOURCES</td>
<td>The request could not be completed due to a lack of resources</td>
</tr>
</tbody>
</table>

**EFI_DISK_IO2_PROTOCOL.FlushDiskEx()**

**Summary**
Flushes all modified data to the physical device.

**Prototype**
```c
typedef
EFI_STATUS
(EIFIAPI *EFI_DISK_FLUSH_EX) (  
    IN EFI_DISK_IO2_PROTOCOL *This,  
    IN OUT EFI_DISK_IO2_TOKEN *Token
);
```

**Parameters**
- **This**
  Indicates a pointer to the calling context. Type **EFI_DISK_IO2_PROTOCOL** is defined in the **EFI_DISK_IO2_PROTOCOL** description.
- **Token**
  A pointer to the token associated with the transaction. Type **EFI_DISK_IO2_TOKEN** is defined in "Related Definitions" below. If this field is NULL, synchronous/blocking I/O is performed.

**Description**
The **FlushDiskEx()** function flushes all modified data to the physical device. If an error is returned from the call to **FlushDiskEx()** and non-blocking I/O is being requested, the **Event** associated with this request will not be signaled. If the call to **FlushDiskEx()** succeeds then the **Event** will be signaled.
upon completion of the flush or if an error occurs during the processing of the request. The status of the flush request can be determined from the Status field of the Token once the event is signaled.

**Status Codes Returned**

<table>
<thead>
<tr>
<th>Code</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>EFI_SUCCESS</td>
<td>Returned from the call FlushDiskEx()</td>
</tr>
<tr>
<td></td>
<td>If Event is NULL (blocking I/O):</td>
</tr>
<tr>
<td></td>
<td>• The data was flushed successfully to the device.</td>
</tr>
<tr>
<td></td>
<td>If Event is not NULL (asynchronous I/O):</td>
</tr>
<tr>
<td></td>
<td>• The request was successfully queued for processing. Event will be signaled upon completion.</td>
</tr>
<tr>
<td></td>
<td>Returned in the token after signaling Event</td>
</tr>
<tr>
<td></td>
<td>The data was flushed successfully to the device.</td>
</tr>
<tr>
<td>EFI_WRITE_PROTECTED</td>
<td>The device cannot be written to.</td>
</tr>
<tr>
<td>EFI_DEVICE_ERROR</td>
<td>The device reported an error while performing the flush operation.</td>
</tr>
<tr>
<td>EFI_NO_MEDIA</td>
<td>There is no medium in the device.</td>
</tr>
<tr>
<td>EFI_MEDIA_CHANGED</td>
<td>The medium in the device has changed since the last access.</td>
</tr>
<tr>
<td>EFI_INVALID_PARAMETER</td>
<td>Token is NULL.</td>
</tr>
<tr>
<td>EFI_OUT_OF_RESOURCES</td>
<td>The request could not be completed due to a lack of resources</td>
</tr>
</tbody>
</table>

**13.9 Block I/O Protocol**

This section defines the Block I/O protocol. This protocol is used to abstract mass storage devices to allow code running in the EFI boot services environment to access them without specific knowledge of the type of device or controller that manages the device. Functions are defined to read and write data at a block level from mass storage devices as well as to manage such devices in the EFI boot services environment.

**EFI_BLOCK_IO_PROTOCOL**

**Summary**

This protocol provides control over block devices.

**GUID**

```c
#define EFI_BLOCK_IO_PROTOCOL_GUID
{0x964e5b21,0x6459,0x11d2,
 {0x8e,0x39,0x00,0xa0,0xc9,0x72,0x3b}}
```

**Revision Number**

```c
#define EFI_BLOCK_IO_PROTOCOL_REVISION2   0x00020001
#define EFI_BLOCK_IO_PROTOCOL_REVISION3   ((2<<16) | (31))
```
Protocol Interface Structure

```c
typedef struct _EFI_BLOCK_IO_PROTOCOL {
    UINT64         Revision;
    EFI_BLOCK_IO_MEDIA *Media;
    EFI_BLOCK_RESET Reset;
    EFI_BLOCK_READ ReadBlocks;
    EFI_BLOCK_WRITE WriteBlocks;
    EFI_BLOCK_FLUSH FlushBlocks;
} EFI_BLOCK_IO_PROTOCOL;
```

Parameters

- **Revision**
  The revision to which the block IO interface adheres. All future revisions must be backwards compatible. If a future version is not backwards compatible it is not the same GUID.

- **Media**
  A pointer to the EFI_BLOCK_IO_MEDIA data for this device. Type EFI_BLOCK_IO_MEDIA is defined in “Related Definitions” below.

- **Reset**
  Resets the block device hardware. See the Reset() function description.

- **ReadBlocks**
  Reads the requested number of blocks from the device. See the ReadBlocks() function description.

- **WriteBlocks**
  Writes the requested number of blocks to the device. See the WriteBlocks() function description.

- **FlushBlocks**
  Flushes any cache blocks. This function is optional and only needs to be supported on block devices that cache writes. See the FlushBlocks() function description.
Related Definitions

//*****************************************************************************
// EFI_BLOCK_IO_MEDIA
//*****************************************************************************

typedef struct {
    UINT32 MediaId;
    BOOLEAN RemovableMedia;
    BOOLEAN MediaPresent;
    BOOLEAN LogicalPartition;
    BOOLEAN ReadOnly;
    BOOLEAN WriteCaching;
    UINT32 BlockSize;
    UINT32 IoAlign;
    EFI_LBA LastBlock;

    EFI_LBA LowestAlignedLba;  //added in Revision 2
    UINT32 LogicalBlocksPerPhysicalBlock;
    //added in Revision 2
    UINT32 OptimalTransferLengthGranularity;
    // added in Revision 3
} EFI_BLOCK_IO_MEDIA;

//*****************************************************************************
// EFI_LBA
//*****************************************************************************
typedef UINT64 EFI_LBA;

The following data values in EFI_BLOCK_IO_MEDIA are read-only and are updated by the code that produces the EFI_BLOCK_IO_PROTOCOL functions:

MediaId The current media ID. If the media changes, this value is changed.
RemovableMedia TRUE if the media is removable; otherwise, FALSE.
MediaPresent TRUE if there is a media currently present in the device; otherwise, FALSE. This field shows the media present status as of the most recent ReadBlocks() or WriteBlocks() call.
LogicalPartition TRUE if the EFI_BLOCK_IO_PROTOCOL was produced to abstract partition structures on the disk. FALSE if the BLOCK_IO protocol was produced to abstract the logical blocks on a hardware device.
ReadOnly TRUE if the media is marked read-only otherwise, FALSE. This field shows the read-only status as of the most recent WriteBlocks() call.
WriteCaching TRUE if the WriteBlocks() function caches write data.
BlockSize The intrinsic block size of the device. If the media changes, then this field is updated. Returns the number of bytes per logical block. For ATA devices, this is reported in IDENTIFY DEVICE data words 117-118.
(i.e., Words per Logical Sector) (see ATA8-ACS). For SCSI devices, this is reported in the READ CAPACITY (16) parameter data Logical Block Length In Bytes field (see SBC-3).

**IoAlign**
Supplies the alignment requirement for any buffer used in a data transfer. **IoAlign** values of 0 and 1 mean that the buffer can be placed anywhere in memory. Otherwise, **IoAlign** must be a power of 2, and the requirement is that the start address of a buffer must be evenly divisible by **IoAlign** with no remainder.

**LastBlock**
The last LBA on the device. If the media changes, then this field is updated. For ATA devices, this is reported in IDENTIFY DEVICE data words 60-61 (i.e., Total number of user addressable logical sectors) (see ATA8-ACS) minus one. For SCSI devices, this is reported in the READ CAPACITY (16) parameter data Returned Logical Block Address field (see SBC-3) minus one.

**LowestAlignedLba**
Only present if **EFI_BLOCK_IO_PROTOCOL.Revision** is greater than or equal to **EFI_BLOCK_IO_PROTOCOL_REVISION2**. Returns the first LBA that is aligned to a physical block boundary (see Section 5.3.1). Note that this field follows the SCSI definition, not the ATA definition. If **LogicalPartition** is **TRUE** this value will be zero.

**LogicalBlocksPerPhysicalBlock**
Only present if **EFI_BLOCK_IO_PROTOCOL.Revision** is greater than or equal to **EFI_BLOCK_IO_PROTOCOL_REVISION2**. Returns the number of logical blocks per physical block (see Section 5.3.1). Unlike the ATA and SCSI fields that provide the information for this field, this field does not contain an exponential value. A value of 0 means there is either one logical block per physical block, or there are more than one physical block per logical block. If **LogicalPartition** is **TRUE** this value will be zero.

**OptimalTransferLengthGranularity**
Only present if **EFI_BLOCK_IO_PROTOCOL.Revision** is greater than or equal to **EFI_BLOCK_IO_PROTOCOL_REVISION3**. Returns the optimal transfer length granularity as a number of logical blocks (see Section 5.3.1). A value of 0 means there is no reported optimal transfer length granularity. If **LogicalPartition** is **TRUE** this value will be zero.

**Description**
The **LogicalPartition** is **TRUE** if the device handle is for a partition. For media that have only one partition, the value will always be **TRUE**. For media that have multiple partitions, this value is **FALSE** for the handle that accesses the entire device. The firmware is responsible for adding device handles for each partition on such media.

The firmware is responsible for adding an **EFI_DISK_IO_PROTOCOL** interface to every **EFI_BLOCK_IO_PROTOCOL** interface in the system. The **EFI_DISK_IO_PROTOCOL** interface allows byte-level access to devices.
**EFI_BLOCK_IO_PROTOCOL.Reset()**

**Summary**
Resets the block device hardware.

**Prototype**
```c
typedef
EFI_STATUS
(EIFIAPI *EFI_BLOCK_RESET) (
  IN EFI_BLOCK_IO_PROTOCOL *This,
  IN BOOLEAN ExtendedVerification
);
```

**Parameters**
- **This** Indicates a pointer to the calling context. Type `EFI_BLOCK_IO_PROTOCOL` is defined in the `EFI_BLOCK_IO_PROTOCOL` description.
- **ExtendedVerification** Indicates that the driver may perform a more exhaustive verification operation of the device during reset.

**Description**
The `Reset()` function resets the block device hardware.

As part of the initialization process, the firmware/device will make a quick but reasonable attempt to verify that the device is functioning. If the `ExtendedVerification` flag is `TRUE` the firmware may take an extended amount of time to verify the device is operating on reset. Otherwise the reset operation is to occur as quickly as possible.

The hardware verification process is not defined by this specification and is left up to the platform firmware or driver to implement.

**Status Codes Returned**

<table>
<thead>
<tr>
<th>Code</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>EFI_SUCCESS</td>
<td>The block device was reset.</td>
</tr>
<tr>
<td>EFI_DEVICE_ERROR</td>
<td>The block device is not functioning correctly and could not be reset.</td>
</tr>
</tbody>
</table>

**EFI_BLOCK_IO_PROTOCOL.ReadBlocks()**

**Summary**
Reads the requested number of blocks from the device.
Prototype

```c
typedef
    EFI_STATUS
    (EFIAPI *EFI_BLOCK_READ) ( 
        IN EFI_BLOCK_IO_PROTOCOL   *This,
        IN UINT32           MediaId,
        IN EFI_LBA           LBA,
        IN UINTN           BufferSize,
        OUT VOID                 *Buffer
    );
```

Parameters

- **This**: Indicates a pointer to the calling context. Type `EFI_BLOCK_IO_PROTOCOL` is defined in the `EFI_BLOCK_IO_PROTOCOL` description.
- **MediaId**: The media ID that the read request is for.
- **LBA**: The starting logical block address to read from on the device. Type `EFI_LBA` is defined in the `EFI_BLOCK_IO_PROTOCOL` description.
- **BufferSize**: The size of the `Buffer` in bytes. This must be a multiple of the intrinsic block size of the device.
- **Buffer**: A pointer to the destination buffer for the data. The caller is responsible for either having implicit or explicit ownership of the buffer.

Description

The `ReadBlocks()` function reads the requested number of blocks from the device. All the blocks are read, or an error is returned.

If there is no media in the device, the function returns `EFI_NO_MEDIA`. If the `MediaId` is not the ID for the current media in the device, the function returns `EFI_MEDIA_CHANGED`. The function must return `EFI_NO_MEDIA` or `EFI_MEDIA_CHANGED` even if `LBA`, `BufferSize`, or `Buffer` are invalid so the caller can probe for changes in media state.

Status Codes Returned

<table>
<thead>
<tr>
<th>Status Code</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>EFI_SUCCESS</td>
<td>The data was read correctly from the device.</td>
</tr>
<tr>
<td>EFI_DEVICE_ERROR</td>
<td>The device reported an error while attempting to perform the read operation.</td>
</tr>
<tr>
<td>EFI_NO_MEDIA</td>
<td>There is no media in the device.</td>
</tr>
<tr>
<td>EFI_MEDIA_CHANGED</td>
<td>The <code>MediaId</code> is not for the current media.</td>
</tr>
<tr>
<td>EFI_BAD_BUFFER_SIZE</td>
<td>The <code>BufferSize</code> parameter is not a multiple of the intrinsic block size of the device.</td>
</tr>
<tr>
<td>EFI_INVALID_PARAMETER</td>
<td>The read request contains LBAs that are not valid, or the buffer is not on proper alignment.</td>
</tr>
</tbody>
</table>
EFI_BLOCK_IO_PROTOCOL.WriteBlocks()

Summary
Writes a specified number of blocks to the device.

Prototype
```c
typedef
    EFI_STATUS
    (EFIAPI *EFI_BLOCK_WRITE) (
        IN EFI_BLOCK_IO_PROTOCOL *This,
        IN UINT32 MediaId,
        IN EFI_LBA LBA,
        IN UINTN BufferSize,
        IN VOID *Buffer
    );
```

Parameters
- **This**: Indicates a pointer to the calling context. Type is defined in the EFI_BLOCK_IO_PROTOCOL description.
- **MediaId**: The media ID that the write request is for.
- **LBA**: The starting logical block address to be written. The caller is responsible for writing to only legitimate locations. Type EFI_LBA is defined in the EFI_BLOCK_IO_PROTOCOL description.
- **BufferSize**: The size in bytes of Buffer. This must be a multiple of the intrinsic block size of the device.
- **Buffer**: A pointer to the source buffer for the data.

Description
The **WriteBlocks()** function writes the requested number of blocks to the device. All blocks are written, or an error is returned.

If there is no media in the device, the function returns **EFI_NO_MEDIA**. If the MediaId is not the ID for the current media in the device, the function returns **EFI_MEDIA_CHANGED**. The function must return **EFI_NO_MEDIA** or **EFI_MEDIA_CHANGED** even if LBA, BufferSize, or Buffer are invalid so the caller can probe for changes in media state.
Status Codes Returned

<table>
<thead>
<tr>
<th>Status Code</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>EFI_SUCCESS</td>
<td>The data were written correctly to the device.</td>
</tr>
<tr>
<td>EFI_WRITE_PROTECTED</td>
<td>The device cannot be written to.</td>
</tr>
<tr>
<td>EFI_NO_MEDIA</td>
<td>There is no media in the device.</td>
</tr>
<tr>
<td>EFI_MEDIA_CHANGED</td>
<td>The MediaId is not for the current media.</td>
</tr>
<tr>
<td>EFI_DEVICE_ERROR</td>
<td>The device reported an error while attempting to perform the write operation.</td>
</tr>
<tr>
<td>EFI_BAD_BUFFER_SIZE</td>
<td>The BufferSize parameter is not a multiple of the intrinsic block size of the device.</td>
</tr>
<tr>
<td>EFI_INVALID_PARAMETER</td>
<td>The write request contains LBAs that are not valid, or the buffer is not on proper alignment.</td>
</tr>
</tbody>
</table>

**EFI_BLOCK_IO_PROTOCOL.FlushBlocks()**

Summary
Flushes all modified data to a physical block device.

Prototype
```c
typedef
EFI_STATUS
(EIFIAPIC *EFI_BLOCK_FLUSH) (  
    IN EFI_BLOCK_IO_PROTOCOL   *This
);
```

Parameters
- **This** Indicates a pointer to the calling context. Type EFI_BLOCK_IO_PROTOCOL is defined in the EFI_BLOCK_IO_PROTOCOL protocol description.

Description
The **FlushBlocks()** function flushes all modified data to the physical block device.

All data written to the device prior to the flush must be physically written before returning EFI_SUCCESS from this function. This would include any cached data the driver may have cached, and cached data the device may have cached. A flush may cause a read request following the flush to force a device access.

Status Codes Returned

<table>
<thead>
<tr>
<th>Status Code</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>EFI_SUCCESS</td>
<td>All outstanding data were written correctly to the device.</td>
</tr>
<tr>
<td>EFI_DEVICE_ERROR</td>
<td>The device reported an error while attempting to write data.</td>
</tr>
<tr>
<td>EFI_NO_MEDIA</td>
<td>There is no media in the device.</td>
</tr>
</tbody>
</table>

**13.10 Block I/O 2 Protocol**

The Block I/O 2 protocol defines an extension to the Block I/O protocol which enables the ability to read and write data at a block level in a non-blocking manner.
**EFI_BLOCK_IO2_PROTOCOL**

**Summary**
This protocol provides control over block devices.

**GUID**
```c
#define EFI_BLOCK_IO2_PROTOCOL_GUID \
{0xa77b2472, 0xe282, 0x4e9f, \ 
 {0xa2, 0x45, 0xc2, 0xc0, 0xe2, 0x7b, 0xbc, 0xc1}}
```

**Protocol Interface Structure**
```c
typedef struct _EFI_BLOCK_IO2_PROTOCOL {
  EFI_BLOCK_IO_MEDIA  *Media;
  EFI_BLOCK_RESET_EX  Reset;
  EFI_BLOCK_READ_EX   ReadBlocksEx;
  EFI_BLOCK_WRITE_EX  WriteBlocksEx;
  EFI_BLOCK_FLUSH_EX  FlushBlocksEx;
} EFI_BLOCK_IO2_PROTOCOL;
```

**Parameters**
- **Media**
  A pointer to the **EFI_BLOCK_IO_MEDIA** data for this device. Type **EFI_BLOCK_IO_MEDIA** is defined in the **EFI_BLOCK_IO_PROTOCOL** section.
- **Reset**
  Resets the block device hardware. See the **Reset()** function description following below.
- **ReadBlocksEx**
  Reads the requested number of blocks from the device. See the **EFI_BLOCK_IO2_PROTOCOL.ReadBlocksEx()** function description.
- **WriteBlocksEx**
  Writes the requested number of blocks to the device. See the **WriteBlocksEx()** function description.
- **FlushBlocksEx**
  Flushes any cache blocks. This function is optional and only needs to be supported on block devices that cache writes. See the **FlushBlocksEx()** function description.

**EFI_BLOCK_IO2_PROTOCOL.Reset()**

**Summary**
Resets the block device hardware.
Prototype

typedef
EFI_STATUS
(EIFIAPIC *EFI_BLOCK_RESET_EX) (
    IN EFI_BLOCK_IO2_PROTOCOL *This,
    IN BOOLEAN ExtendedVerification
);

Parameters

This Indicates a pointer to the calling context. Type
EFI_BLOCK_IO2_PROTOCOL is defined in the
EFI_BLOCK_IO2_PROTOCOL description.

ExtendedVerification Indicates that the driver may perform a more exhaustive verification
operation of the device during reset.

Description

The Reset() function resets the block device hardware.

As part of the initialization process, the firmware/device will make a quick but reasonable attempt to
verify that the device is functioning. If the ExtendedVerification flag is TRUE the firmware may take
an extended amount of time to verify the device is operating on reset. Otherwise the reset operation is to
occur as quickly as possible.

The hardware verification process is not defined by this specification and is left up to the platform
firmware or driver to implement.

The Reset() function will terminate any in-flight non-blocking I/O requests by signaling an
EFI_ABORTED in the TransactionStatus member of the EFI_BLOCK_IO2_TOKEN for the non-
blocking I/O. After the Reset() function returns it is safe to free any Token or Buffer data structures
that were allocated to initiate the non-blocking I/O requests that were in-flight for this device.

Status Codes Returned

<table>
<thead>
<tr>
<th>Code</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>EFI_SUCCESS</td>
<td>The block device was reset.</td>
</tr>
<tr>
<td>EFI_DEVICE_ERROR</td>
<td>The block device is not functioning correctly and could not be reset.</td>
</tr>
</tbody>
</table>

EFI_BLOCK_IO2_PROTOCOL.ReadBlocksEx()

Summary

Reads the requested number of blocks from the device.
Prototype

typedef EFI_STATUS (EFIAPI *EFI_BLOCK_READ_EX) (
    IN EFI_BLOCK_IO2_PROTOCOL *This,
    IN UINT32 MediaId,
    IN EFI_LBA LBA,
    IN OUT EFI_BLOCK_IO2_TOKEN *Token,
    IN UINTN BufferSize,
    OUT VOID *Buffer
);

Parameters

This Indicates a pointer to the calling context. Type EFI_BLOCK_IO2_PROTOCOL is defined in the EFI_BLOCK_IO2_PROTOCOL description.

MediaId The media ID that the read request is for.

LBA The starting logical block address to read from on the device. Type EFI_LBA is defined in the EFI_BLOCK_IO_PROTOCOL description.

Token A pointer to the token associated with the transaction. Type EFI_BLOCK_IO2_TOKEN is defined in "Related Definitions" below.

BufferSize The size of the Buffer in bytes. This must be a multiple of the intrinsic block size of the device.

Buffer A pointer to the destination buffer for the data. The caller is responsible for either having implicit or explicit ownership of the buffer.

Description

The ReadBlocksEx() function reads the requested number of blocks from the device. All the blocks are read, or an error is returned.

If there is no media in the device, the function returns EFI_NO_MEDIA. If the MediaId is not the ID for the current media in the device, the function returns EFI_MEDIA_CHANGED. The function must return EFI_NO_MEDIA or EFI_MEDIA_CHANGED even if LBA, BufferSize, or Buffer are invalid so the caller can probe for changes in media state.

If EFI_DEVICE_ERROR, EFI_NO_MEDIA, or EFI_MEDIA_CHANGED is returned and non-blocking I/O is being used, the Event associated with this request will not be signaled.

Related Definitions

typedef struct {
    EFIEVENT Event;
    EFI_STATUS TransactionStatus;
} EFI_BLOCK_IO2_TOKEN;
Event

If Event is NULL, then blocking I/O is performed. If Event is not NULL and non-blocking I/O is supported, then non-blocking I/O is performed, and Event will be signaled when the read request is completed.

TransactionStatus

Defines whether the signaled event encountered an error.

Status Codes Returned

<table>
<thead>
<tr>
<th>Code</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>EFI_SUCCESS</td>
<td>The read request was queued if Token-&gt;Event is not NULL. The data was read correctly from the device if theToken-&gt;Event is NULL.</td>
</tr>
<tr>
<td>EFI_DEVICE_ERROR</td>
<td>The device reported an error while attempting to perform the read operation.</td>
</tr>
<tr>
<td>EFI_NO_MEDIA</td>
<td>There is no media in the device.</td>
</tr>
<tr>
<td>EFI_MEDIA_CHANGED</td>
<td>The MediaId is not for the current media.</td>
</tr>
<tr>
<td>EFI_BAD_BUFFER_SIZE</td>
<td>The BufferSize parameter is not a multiple of the intrinsic block size of the device.</td>
</tr>
<tr>
<td>EFI_INVALID_PARAMETER</td>
<td>The read request contains LBAs that are not valid, or the buffer is not on proper alignment.</td>
</tr>
<tr>
<td>EFI_OUT_OF_RESOURCES</td>
<td>The request could not be completed due to a lack of resources</td>
</tr>
</tbody>
</table>

**EFI_BLOCK_IO2_PROTOCOL.WriteBlocksEx()**

**Summary**

Writes a specified number of blocks to the device.

**Prototype**

```c
typedef EFI_STATUS
(EFIAPI *EFI_BLOCK_WRITE_EX) (  
    IN EFI_BLOCK_IO2_PROTOCOL *This,  
    IN UINT32 MediaId,  
    IN EFI_LBA LBA,  
    IN OUT EFI_BLOCK_IO2_TOKEN *Token,  
    IN UINTN BufferSize,  
    IN VOID *Buffer  
);
```

**Parameters**

- **This**
  Indicates a pointer to the calling context. Type **EFI_BLOCK_IO2_PROTOCOL** is defined in the **EFI_BLOCK_IO2_PROTOCOL** description.

- **MediaId**
  The media ID that the write request is for.

- **LBA**
  The starting logical block address to be written. The caller is responsible for writing to only legitimate locations. Type **EFI_LBA** is defined in the **EFI_BLOCK_IO2_PROTOCOL** description.
Token
A pointer to the token associated with the transaction. Type
EFI_BLOCK_IO2_TOKEN is defined in
EFI_BLOCK_IO2_PROTOCOL.ReadBlocksEx(). "Related
Definitions".

BufferSize
The size in bytes of Buffer. This must be a multiple of the intrinsic
block size of the device.

Buffer
A pointer to the source buffer for the data.

Description
The WriteBlocksEx() function writes the requested number of blocks to the device. All blocks are written,
or an error is returned.

If there is no media in the device, the function returns EFI_NO_MEDIA. If the MediaId is not the ID for the
current media in the device, the function returns EFI_MEDIA_CHANGED. The function must return
EFI_NO_MEDIA or EFI_MEDIA_CHANGED even if LBA, BufferSize, or Buffer are invalid so the caller
can probe for changes in media state.

If EFI_DEVICE_ERROR, EFI_NO_MEDIA, EFI_WRITE_PROTECTED or EFI_MEDIA_CHANGED is
returned and non-blocking I/O is being used, the Event associated with this request will not be signaled.

Related Definitions
typedef struct {
  EFI_EVENT      Event;
  EFI_STATUS     TransactionStatus;
} EFI_BLOCK_IO2_TOKEN;

Event
If Event is NULL, then blocking I/O is performed. If Event is not
NULL and non-blocking I/O is supported, then non-blocking I/O is
performed, and Event will be signaled when the write request is
completed.

TransactionStatus
Defines whether the signaled event encountered an error.
Status Codes Returned

<table>
<thead>
<tr>
<th>Code</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>EFI_SUCCESS</td>
<td>The write request was queued if Event is not NULL. The data was written</td>
</tr>
<tr>
<td></td>
<td>correctly to the device if the Event is NULL.</td>
</tr>
<tr>
<td>EFI_WRITE_PROTECTED</td>
<td>The device cannot be written to.</td>
</tr>
<tr>
<td>EFI_NO_MEDIA</td>
<td>There is no media in the device.</td>
</tr>
<tr>
<td>EFI_MEDIA_CHANGED</td>
<td>The MediaId is not for the current media.</td>
</tr>
<tr>
<td>EFI_DEVICE_ERROR</td>
<td>The device reported an error while attempting to perform the write</td>
</tr>
<tr>
<td></td>
<td>operation.</td>
</tr>
<tr>
<td>EFI_BAD_BUFFER_SIZE</td>
<td>The BufferSize parameter is not a multiple of the intrinsic block size of</td>
</tr>
<tr>
<td></td>
<td>the device.</td>
</tr>
<tr>
<td>EFI_INVALID_PARAMETER</td>
<td>The write request contains LBAs that are not valid, or the buffer is not on</td>
</tr>
<tr>
<td></td>
<td>proper alignment.</td>
</tr>
<tr>
<td>EFI_OUT_OF_RESOURCES</td>
<td>The request could not be completed due to a lack of resources.</td>
</tr>
</tbody>
</table>

**EFI_BLOCK_IO2_PROTOCOL.FlushBlocksEx()**

**Summary**
Flushes all modified data to a physical block device.

**Prototype**
```c
typedef EFI_STATUS (EFIAPI *EFI_BLOCK_FLUSH_EX) (  
    IN EFI_BLOCK_IO2_PROTOCOL *This,  
    IN OUT EFI_BLOCK_IO2_TOKEN *Token,  
);
```

**Parameters**
- **This** Indicates a pointer to the calling context. Type     
  **EFI_BLOCK_IO2_PROTOCOL** is defined in the     
  **EFI_BLOCK_IO2_PROTOCOL** protocol description.  
- **Token** A pointer to the token associated with the transaction. Type     
  **EFI_BLOCK_IO2_TOKEN** is defined in     
  **EFI_BLOCK_IO2_PROTOCOL.ReadBlocksEx()**, "Related     
  Definitions".

**Related Definitions**
```c
typedef struct {    
    EFI_EVENT Event;    
    EFI_STATUS TransactionStatus;    
} EFI_BLOCK_IO2_TOKEN;
```
**Event**

If `Event` is NULL, then blocking I/O is performed. If `Event` is not NULL and non-blocking I/O is supported, then non-blocking I/O is performed, and `Event` will be signaled when the flush request is completed.

**TransactionStatus**

Defines whether the signaled event encountered an error.

**Description**

The `FlushBlocksEx()` function flushes all modified data to the physical block device.

All data written to the device prior to the flush must be physically written before returning `EFI_SUCCESS` from this function. This would include any cached data the driver may have cached, and cached data the device may have cached. A flush may cause a read request following the flush to force a device access.

If `EFI_DEVICE_ERROR, EFI_NO_MEDIA, EFI_WRITE_PROTECTED` or `EFI_MEDIA_CHANGED` is returned and non-blocking I/O is being used, the `Event` associated with this request will not be signaled.

**Status Codes Returned**

<table>
<thead>
<tr>
<th>Code</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>EFI_SUCCESS</td>
<td>The flush request was queued if <code>Event</code> is not NULL. All outstanding data was written correctly to the device if the <code>Event</code> is NULL.</td>
</tr>
<tr>
<td>EFI_DEVICE_ERROR</td>
<td>The device reported an error while attempting to write data.</td>
</tr>
<tr>
<td>EFI_WRITE_PROTECTED</td>
<td>The device cannot be written to.</td>
</tr>
<tr>
<td>EFI_NO_MEDIA</td>
<td>There is no media in the device.</td>
</tr>
<tr>
<td>EFI_MEDIA_CHANGED</td>
<td>The <code>MediaId</code> is not for the current media.</td>
</tr>
<tr>
<td>EFI_OUT_OF_RESOURCES</td>
<td>The request could not be completed due to a lack of resources.</td>
</tr>
</tbody>
</table>

**13.11 Inline Cryptographic Interface Protocol**

**EFI_BLOCK_IO_CRYPTO_PROTOCOL**

**Summary**

The UEFI Inline Cryptographic Interface protocol provides services to abstract access to inline cryptographic capabilities.

The usage model of this protocol is similar to the one of the `EFI_STORAGE_SECURITY_COMMAND_PROTOCOL` where FDE (Full Disk Encryption) solutions leave ESP partition unprotected (unencrypted) allowing storage clients to continue using `EFI_BLOCK_IO_PROTOCOL` or `EFI_BLOCK_IO2_PROTOCOL` protocol interfaces to load OS boot components from ESP partition. For other partitions boot apps (including OS boot app) that are enlightened to take advantage of inline cryptographic capability will be empowered to use this new protocol.
GUID

```c
#define EFI_BLOCK_IO_CRYPTO_PROTOCOL_GUID \
{0xa00490ba,0x3f1a,0x4b4c,\ 
{0xab,0x90,0x4f,0xa9,0x97,0x26,0xa1,0xe8}}
```

Protocol Interface Structure

```c
typedef struct _EFI_BLOCK_IO_CRYPTO_PROTOCOL {
    EFI_BLOCK_IO_MEDIA *Media;
    EFI_BLOCK_IO_CRYPTO_RESET Reset;
    EFI_BLOCK_IO_CRYPTO_GET_CAPABILITIES GetCapabilities;
    EFI_BLOCK_IO_CRYPTO_SET_CONFIGURATION SetConfiguration;
    EFI_BLOCK_IO_CRYPTO_GET_CONFIGURATION GetConfiguration;
    EFI_BLOCK_IO_CRYPTO_READ_DEVICE_EXTENDED ReadExtended;
    EFI_BLOCK_IO_CRYPTO_WRITEDEVICE_EXTENDED WriteExtended;
    EFI_BLOCK_IO_CRYPTO_FLUSH FlushBlocks;
} EFI_BLOCK_IO_CRYPTO_PROTOCOL;
```

Parameters

- **Media**
  A pointer to the `EFI_BLOCK_IO_MEDIA` data for this device. Type `EFI_BLOCK_IO_MEDIA` is defined in the `EFI_BLOCK_IO_PROTOCOL` section.
- **Reset**
  Reset the block device hardware.
- **GetCapabilities**
  Get the current capabilities of the ICI.
- **SetConfiguration**
  Set the configuration for the ICI instance.
- **GetConfiguration**
  Get the configuration for the ICI instance.
- **ReadExtended**
  Provide an extended version of the storage device read command.
- **WriteExtended**
  Provide an extended version of the storage device write command.
- **FlushBlocks**
  Flushes any cache blocks. This function is optional and only needs to be supported on block devices that cache writes.

Related Definitions

Some functions defined for this protocol require the caller to specify the device capabilities, keys and/or attributes of the keys to be used. These parameters must be consistent with the supported capabilities as reported by the device.

```c
typedef struct {
    EFI_GUID Algorithm;
    UINT64 KeySize;
    UINT64 CryptoBlockSizeBitMask;
} EFI_BLOCK_IO_CRYPTO_CAPABILITY;
```

- **Algorithm**
  GUID of the algorithm.
- **KeySize**
  Specifies `KeySize` in bits used with this Algorithm.
- **CryptoBlockSizeBitMask**
Specifies bitmask of block sizes supported by this algorithm. Bit \( j \) being set means that \( 2^j \) bytes crypto block size is supported.

```c
#define EFI_BLOCK_IO_CRYPTO_ALGO_GUID_AES_XTS 
  {0x2f87ba6a, 
   0x5c04,0x4385,0xa7,0x80,0xf3,0xbf,0x78,0xa9,0x7b,0xec}
```

`EFI_BLOCK_IO_CRYPTO_ALGO_GUID_AES_XTS` GUID represents Inline Cryptographic Interface capability supporting AES XTS crypto algorithm as described in IEEE Std 1619-2007: IEEE Standard for Cryptographic Protection of Data on Block-Oriented Storage Devices.

```c
typedef struct { 
  EFI_BLOCK_IO_CRYPTO_IV_INPUT Header; 
  UINT64 CryptoBlockNumber; 
  UINT64 CryptoBlockByteSize; 
} EFI_BLOCK_IO_CRYPTO_IV_INPUT_AES_XTS;
```

`EFI_BLOCK_IO_CRYPTO_IV_INPUT_AES_XTS` structure is used as `CryptoIvInput` parameter to the `ReadExtended` and `WriteExtended` methods for Inline Cryptographic Interface supporting and using AES XTS algorithm with IV input as defined for AES XTS algorithm. IO operation (read or write) range should consist of one or more blocks of `CryptoBlockByteSize` size. `CryptoBlockNumber` is used as the AES XTS IV for the first crypto block and is incremented by one for each consecutive crypto block in the IO operation range.

```c
#define EFI_BLOCK_IO_CRYPTO_ALGO_GUID_AES_CBC_MICROSOFT_BITLOCKER 
  {0x689e4c62, 
   0x70bf,0x4cf3,0x88,0xbb,0x33,0xb3,0x18,0x26,0x86,0x70}
```

`EFI_BLOCK_IO_CRYPTO_ALGO_GUID_AES_CBC_MICROSOFT_BITLOCKER` GUID represents Inline Cryptographic Interface capability supporting AES CBC crypto algorithm in the non-diffuser mode as described in following Microsoft white paper, section 4: See “Links to UEFI-Related Documents” ([http://uefi.org/uefi](http://uefi.org/uefi)) under the heading “Inline Cryptographic Interface--Bit Locker Cipher”. It is important to note that when excluding diffuser operations (A diffuser and B diffuser) described in the above document one should also exclude derivation of sector key and XOR-ing it with plaintext as that operation is part of the diffuser part of the algorithm and does not belong to the AES-CBC Microsoft BitLocker algorithm being referred to here.

```c
typedef struct { 
  EFI_BLOCK_IO_CRYPTO_IV_INPUT Header; 
  UINT64 CryptoBlockByteOffset; 
  UINT64 CryptoBlockByteSize; 
} EFI_BLOCK_IO_CRYPTO_IV_INPUT_AES_CBC_MICROSOFT_BITLOCKER;
```

`EFI_BLOCK_IO_CRYPTO_IV_INPUT_AES_CBC_MICROSOFT_BITLOCKER` structure is used to pass as `CryptoIvInput` parameter to the `ReadExtended` and `WriteExtended` methods for Inline Cryptographic Interface supporting and using AES CBC algorithm with IV input as defined for Microsoft BitLocker Drive Encryption. IO operation (read or write) range should consist of one or more blocks of `CryptoBlockByteSize` size. `CryptoBlockByteOffset` is used as the AES CBC Microsoft BitLocker algorithm IV for the first crypto block and is incremented by `CryptoBlockByteSize` for each consecutive crypto block in the IO operation range.
typedef struct {
    UINT64 InputSize;
} EFI_BLOCK_IO_CRYPTO_IV_INPUT;

EFI_BLOCK_IO_CRYPTO_IV_INPUT structure is used as a common header in CryptoIvInput parameters passed to the ReadExtended and WriteExtended methods for Inline Cryptographic Interface. Its purpose is to pass size of the entire CryptoIvInput parameter memory buffer to the Inline Cryptographic Interface.

Further extensions of crypto algorithm support by Inline Cryptographic Interface should follow the same pattern established above for the AES XTS and AES CBC Microsoft BitLocker algorithms. In particular each added crypto algorithm should:

- Define its crypto algorithm GUID using following pattern:
  #define EFI_BLOCK_IO_CRYPTO_ALGO_GUID_<algo-name> {<algo-guid>}
- Define its corresponding CryptoIvInput parameter structure and describe how it is populated for each IO operation (read / write):

typedef struct {
    EFI_BLOCK_IO_CRYPTO_IV_INPUT Header;
    <TBD> <TBD>;
} EFI_BLOCK_IO_CRYPTO_IV_INPUT_<algo-name>;

#define EFI_BLOCK_IO_CRYPTO_INDEX_ANY 0xFFFFFFFFFFFFFFFF

typedef struct {
    BOOLEAN Supported;
    UINT64 KeyCount;
    UINT64 CapabilityCount;
    EFI_BLOCK_IO_CRYPTO_CAPABILITY Capabilities[1];
} EFI_BLOCK_IO_CRYPTO_CAPABILITIES;

Supported Is inline cryptographic capability supported on this device.
KeyCount Maximum number of keys that can be configured at the same time.
CapabilityCount Number of supported capabilities.
Capabilities Array of supported capabilities.

typedef struct {
    UINT64 Index;
    EFI_GUID KeyOwnerGuid;
    EFI_BLOCK_IO_CRYPTO_CAPABILITY Capability;
    VOID *CryptoKey;
} EFI_BLOCK_IO_CRYPTO_CONFIGURATION_TABLE_ENTRY;
**Index**  
Configuration table index. A special Index **EFI_BLOCK_IO_CRYPTO_INDEX_ANY** can be used to set any available entry in the configuration table.

**KeyOwnerGuid**  
Identifies the owner of the configuration table entry. Entry can also be used with the Nil value to clear key from the configuration table index.

**Capability**  
A supported capability to be used. The **CryptoBlockSizeBitMask** field of the structure should have only one bit set from the supported mask.

**CryptoKey**  
Pointer to the key. The size of the key is defined by the **KeySize** field of the capability specified by the Capability parameter.

```c
typedef struct {
    UINT64                         Index;
    EFI_GUID                      KeyOwnerGuid;
    EFI_BLOCK_IO_CRYPTO_CAPABILITY Capability;
} EFI_BLOCK_IO_CRYPTO_RESPONSE_CONFIGURATION_ENTRY;
```

**Index**  
Configuration table index.

**KeyOwnerGuid**  
Identifies the current owner of the entry.

**Capability**  
The capability to be used. The **CryptoBlockSizeBitMask** field of the structure has only one bit set from the supported mask.

**Description**
The **EFI_BLOCK_IO_CRYPTO_PROTOCOL** defines a UEFI protocol that can be used by UEFI drivers and applications to perform block encryption on a storage device, such as UFS.

The **EFI_BLOCK_IO_CRYPTO_PROTOCOL** instance will be on the same handle as the device path of the inline encryption device.

While this protocol is intended to abstract the encryption process for block device access, the protocol user does not have to be aware of the specific underlying encryption hardware.

**EFI_BLOCK_IO_CRYPTO_PROTOCOL.Reset()**

**Summary**
Resets the block device hardware.

**Prototype**
```c
typedef EFI_STATUS
(EFI_API *EFI_BLOCK_IO_CRYPTO_RESET)(
    IN EFI_BLOCK_IO_CRYPTO_PROTOCOL  *This,
    IN BOOLEAN                      ExtendedVerification
);```
Parameters

This Pointer to the **EFI_BLOCK_IO_CRYPTO_PROTOCOL** instance.

ExtendedVerification Indicates that the driver may perform a more exhaustive verification operation of the device during reset.

Description

The **Reset()** function resets the block device hardware.

As part of the initialization process, the firmware/device will make a quick but reasonable attempt to verify that the device is functioning. If the **ExtendedVerification** flag is TRUE the firmware may take an extended amount of time to verify the device is operating on reset. Otherwise the reset operation is to occur as quickly as possible.

The hardware verification process is not defined by this specification and is left up to the platform firmware or driver to implement.

**Status Codes Returned**

- **EFI_SUCCESS** The block device was reset.
- **EFI_DEVICE_ERROR** The block device is not functioning correctly and could not be reset.
- **EFI_INVALID_PARAMETER** This is NULL.

**EFI_BLOCK_IO_CRYPTO_PROTOCOL.GetCapabilities()**

**Summary**

Get the capabilities of the underlying inline cryptographic interface.

**Prototype**

```c
typedef EFI_STATUS
  (EFIAPI *EFI_BLOCK_IO_CRYPTO_GET_CAPABILITIES) ( 
    IN EFI_BLOCK_IO_CRYPTO_PROTOCOL *This,
    OUT EFI_BLOCK_IO_CRYPTO_CAPABILITIES *Capabilities
  );
```

**Parameters**

- **This** Pointer to the **EFI_BLOCK_IO_CRYPTO_PROTOCOL** instance.
- **Capabilities** Pointer to the **EFI_BLOCK_IO_CRYPTO_CAPABILITIES** structure.

**Description**

The **GetCapabilities()** function determines whether pre-OS controllable inline crypto is supported by the system for the current disk and, if so, returns the capabilities of the crypto engine.
The caller is responsible for providing the Capabilities structure with a sufficient number of entries. If the structure is too small, the EFI_BUFFER_TOO_SMALL error code is returned and the CapabilityCount field contains the number of entries needed to contain the capabilities.

### Status Codes Returned

<table>
<thead>
<tr>
<th>Status Code</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>EFI_SUCCESS</td>
<td>The ICI is ready for use.</td>
</tr>
<tr>
<td>EFI_BUFFER_TOO_SMALL</td>
<td>The Capabilities structure was too small. The number of entries needed is returned in the CapabilityCount field of the structure.</td>
</tr>
<tr>
<td>EFI_NO_RESPONSE</td>
<td>No response was received from the ICI.</td>
</tr>
<tr>
<td>EFI_DEVICE_ERROR</td>
<td>An error occurred when attempting to access the ICI.</td>
</tr>
<tr>
<td>EFI_INVALID_PARAMETER</td>
<td>This is NULL.</td>
</tr>
<tr>
<td>EFI_INVALID_PARAMETER</td>
<td>Capabilities is NULL.</td>
</tr>
</tbody>
</table>

### EFI_BLOCK_IO_CRYPTO_PROTOCOL.SetConfiguration()

**Summary**

Set the configuration of the underlying inline cryptographic interface.

**Prototype**

```c
typedef EFI_STATUS
(EFIAPI *EFI_BLOCK_IO_CRYPTO_PROTOCOL) (  
    IN EFI_BLOCK_IO_CRYPTO_PROTOCOL *This,  
    IN UINT64 ConfigurationCount,  
    IN EFI_BLOCK_IO_CRYPTO_CONFIGURATION_TABLE_ENTRY *ConfigurationTable,  
    OUT EFI_BLOCK_IO_CRYPTO_RESPONSEConfigurationException Entry *ResultingTable OPTIONAL  
);  
```

**Parameters**

- **This** Pointer to the EFI_BLOCK_IO_CRYPTO_PROTOCOL instance.
- **ConfigurationCount** Number of entries being configured with this call.
- **ConfigurationTable** Pointer to a table used to populate the configuration table.
- **ResultingTable** Optional pointer to a table that receives the newly configured entries.

**Description**

The SetConfiguration() function allows the user to set the current configuration of the inline cryptographic interface and should be called before attempting any crypto operations.

This configures the configuration table entries with algorithms, key sizes and keys. Each configured entry can later be referred to by index at the time of storage transaction.

The configuration table index will refer to the combination of KeyOwnerGuid, Algorithm, and CryptoKey.
KeyOwnerGuid identifies the component taking ownership of the entry. It helps components to identify their own entries, cooperate with other owner components, and avoid conflicts. This Guid identifier is there to help coordination between cooperating components and not a security or synchronization feature. The Nil GUID can be used by a component to release use of entry owned. It is also used to identify potentially available entries (see GetConfiguration).

CryptoKey specifies algorithm-specific key material to use within parameters of selected crypto capability.

This function is called infrequently – typically once, on device start, before IO starts. It can be called at later times in cases the number of keys used on the drive is higher than what can be configured at a time or a new key has to be added.

Components setting or changing an entry or entries for a given index or indices must ensure that IO referencing affected indices is temporarily blocked (run-down) at the time of change.

Indices parameters in each parameter table entry allow to set only a portion of the available table entries in the crypto module anywhere from single entry to entire table supported.

If corresponding table entry or entries being set are already in use by another owner the call should be failed and none of the entries should be modified. The interface implementation must enforce atomicity of this operation (should either succeed fully or fail completely without modifying state). Note that components using GetConfiguration command to discover available entries should be prepared that by the time of calling SetConfiguration the previously available entry may have become occupied. Such components should be prepared to re-try the sequence of operations. Alternatively EFIBLOCK_IO_CRYPTO_INDEX_ANY can be used to have the implementation discover and allocate available, if any, indices atomically.

An optional ResultingTable pointer can be provided by the caller to receive the newly configured entries. The array provided by the caller must have at least ConfigurationCount of entries.

### Status Codes Returned

<table>
<thead>
<tr>
<th>Status Code</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>EFI_SUCCESS</td>
<td>The ICI is ready for use.</td>
</tr>
<tr>
<td>EFI_NO_RESPONSE</td>
<td>No response was received from the ICI</td>
</tr>
<tr>
<td>EFI_DEVICE_ERROR</td>
<td>An error occurred when attempting to access the ICI</td>
</tr>
<tr>
<td>EFI_INVALID_PARAMETER</td>
<td>This is NULL.</td>
</tr>
<tr>
<td>EFI_INVALID_PARAMETER</td>
<td>ConfigurationTable is NULL</td>
</tr>
<tr>
<td>EFI_INVALID_PARAMETER</td>
<td>ConfigurationCount is 0</td>
</tr>
<tr>
<td>EFI_OUT_OF_RESOURCES</td>
<td>Could not find the requested number of available entries in the configuration table.</td>
</tr>
</tbody>
</table>
**EFI_BLOCK_IO_CRYPTO_PROTOCOL.GetConfiguration()**

**Summary**
Get the configuration of the underlying inline cryptographic interface.

**Prototype**
```
typedef EFI_STATUS
  (EFIAPI *EFI_BLOCK_IO_CRYPTO_GET_CONFIGURATION) (
   IN EFI_BLOCK_IO_CRYPTO_PROTOCOL *This,
   IN UINT64 StartIndex,
   IN UINT64 ConfigurationCount,
   IN EFI_GUID *KeyOwnerGuid OPTIONAL,
   OUT EFI_BLOCK_IO_CRYPTO_RESPONSE_CONFIGURATION_ENTRY *ConfigurationTable
  );
```

**Parameters**
- **This**: Pointer to the `EFI_BLOCK_IO_CRYPTO_PROTOCOL` instance.
- **StartIndex**: Configuration table index at which to start the configuration query.
- **ConfigurationCount**: Number of entries to return in the response table.
- **KeyOwnerGuid**: Optional parameter to filter response down to entries with a given owner. A pointer to the Nil value can be used to return available entries. Set to NULL when no owner filtering is required.
- **ConfigurationTable**: Table of configured configuration table entries (with no CryptoKey returned): configuration table index, `KeyOwnerGuid`, Capability. Should have sufficient space to store up to `ConfigurationCount` entries.

**Description**
The `GetConfiguration()` function allows the user to get the configuration of the inline cryptographic interface.

Retrieves, entirely or partially, the currently configured key table. Note that the keys themselves are not retrieved, but rather just indices, owner GUIDs and capabilities.

If fewer entries than specified by `ConfigurationCount` are returned, the Index field of the unused entries is set to `EFI_BLOCK_IO_CRYPTO_INDEX_ANY`. 
Status Codes Returned

<table>
<thead>
<tr>
<th>Code</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>EFI_SUCCESS</td>
<td>The ICI is ready for use.</td>
</tr>
<tr>
<td>EFI_NO_RESPONSE</td>
<td>No response was received from the ICI</td>
</tr>
<tr>
<td>EFI_DEVICE_ERROR</td>
<td>An error occurred when attempting to access the ICI</td>
</tr>
<tr>
<td>EFI_INVALID_PARAMETER</td>
<td>This is NULL.</td>
</tr>
<tr>
<td>EFI_INVALID_PARAMETER</td>
<td>Configuration table is NULL</td>
</tr>
<tr>
<td>EFI_INVALID_PARAMETER</td>
<td>StartIndex is out of bounds</td>
</tr>
</tbody>
</table>

**EFI_BLOCK_IO_CRYPTO_PROTOCOL.ReadExtended()**

**Summary**

Reads the requested number of blocks from the device and optionally decrypts them inline.

**Prototype**

```c
typedef EFI_STATUS
  (EFIAPI *EFI_BLOCK_IO_CRYPTO_READ_EXTENDED) (
  IN EFI_BLOCK_IO_CRYPTO_PROTOCOL   *This,
  IN UINT32                         MediaId,
  IN EFI_LBA                        LBA,
  IN OUT EFI_BLOCK_IO_CRYPTO_TOKEN  *Token,
  IN UINT64                         BufferSize,
  OUT VOID                          *Buffer,
  IN UINT64                         *Index OPTIONAL,
  IN VOID                           *CryptoIvInput OPTIONAL
);
```

**Parameters**

- **This**: Pointer to the `EFI_BLOCK_IO_CRYPTO_PROTOCOL` instance.
- **MediaId**: The media ID that the read request is for.
- **LBA**: The starting logical block address to read from on the device. Type `EFI_LBA` is defined in the `EFI_BLOCK_IO_PROTOCOL` description.
- **Token**: A pointer to the token associated with the transaction. Type `EFI_BLOCK_IO_CRYPTO_TOKEN` is defined in “Related Definitions” below.
- **BufferSize**: The size of the `Buffer` in bytes. This must be a multiple of the intrinsic block size of the device.
- **Buffer**: A pointer to the destination buffer for the data. The caller is responsible for either having implicit or explicit ownership of the buffer.
- **Index**: A pointer to the configuration table index. This is optional.
- **CryptoIvInput**: A pointer to a buffer that contains additional cryptographic parameters as required by the capability referenced by the configuration table index, such as cryptographic initialization vector.
Description

The `ReadExtended()` function allows the caller to perform a storage device read operation. The function reads the requested number of blocks from the device and then if `Index` is specified decrypts them inline. All the blocks are read and decrypted (if decryption requested), or an error is returned.

If there is no media in the device, the function returns `EFI_NO_MEDIA`. If the `MediaId` is not the ID for the current media in the device, the function returns `EFI_MEDIA_CHANGED`.

If `EFI_DEVICE_ERROR`, `EFI_NO_MEDIA`, or `EFI_MEDIA_CHANGED` is returned and non-blocking I/O is being used, the `Event` associated with this request will not be signaled.

In addition to standard storage transaction parameters (LBA, IO size, and buffer), this command will also specify a configuration table `Index` and `CryptoIvInput` when data has to be decrypted inline by the controller after being read from the storage device. If an `Index` parameter is not specified, no decryption is performed.

Related Definitions

```c
typedef struct {
  EFI_EVENT Event;
  EFI_STATUS TransactionStatus;
} EFI_BLOCK_IO_CRYPTO_TOKEN;
```

- **Event**
  
  If `Event` is NULL, then blocking I/O is performed. If `Event` is not NULL and non-blocking I/O is supported, then non-blocking I/O is performed, and `Event` will be signaled when the read request is completed and data was decrypted (when `Index` was specified).

- **TransactionStatus**
  
  Defines whether or not the signaled event encountered an error.
Status Codes Returned

<table>
<thead>
<tr>
<th>Status Code</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>EFI_SUCCESS</td>
<td>The read request was queued if Token-&gt;Event is not NULL. The data was read correctly from the device if the Token-&gt;Event is NULL.</td>
</tr>
<tr>
<td>EFI_DEVICE_ERROR</td>
<td>The device reported an error while attempting to perform the read operation and/or decryption operation.</td>
</tr>
<tr>
<td>EFI_NO_MEDIA</td>
<td>There is no media in the device.</td>
</tr>
<tr>
<td>EFI_MEDIA_CHANGED</td>
<td>The MediaId is not for the current media.</td>
</tr>
<tr>
<td>EFI_BAD_BUFFER_SIZE</td>
<td>The BufferSize parameter is not a multiple of the intrinsic block size of the device.</td>
</tr>
<tr>
<td>EFI_INVALID_PARAMETER</td>
<td>This is NULL, or the read request contains LBAs that are not valid, or the buffer is not on proper alignment.</td>
</tr>
<tr>
<td>EFI_INVALID_PARAMETER</td>
<td>CryptoIvInput is incorrect.</td>
</tr>
<tr>
<td>EFI_OUT_OF_RESOURCES</td>
<td>The request could not be completed due to a lack of resources.</td>
</tr>
</tbody>
</table>

**EFI_BLOCK_IO_CRYPTO_PROTOCOL.WriteExtended()**

**Summary**
Optionally encrypts a specified number of blocks inline and then writes to the device.

**Prototype**
```c
typedef EFI_STATUS
(EFIAPI *EFI_BLOCK_IO_CRYPTO_WRITE_EXTENDED) (  
  IN EFI_BLOCK_IO_CRYPTO_PROTOCOL *This,  
  IN UINT32 MediaId,  
  IN EFI_LBA LBA,  
  IN OUT EFI_BLOCK_IO_CRYPTO_TOKEN *Token,  
  IN UINT64 BufferSize,  
  IN VOID *Buffer,  
  IN UINT64 *Index, OPTIONAL  
  *CryptoIvInput OPTIONAL  
);
```

**Parameters**
- **This** — Pointer to the EFI_BLOCK_IO_CRYPTO_PROTOCOL instance.
- **MediaId** — The media ID that the read request is for.
- **LBA** — The starting logical block address to read from on the device. Type EFI_LBA is defined in the EFI_BLOCK_IO_PROTOCOL description.
- **Token** — A pointer to the token associated with the transaction. Type EFI_BLOCK_IO_CRYPTO_TOKEN is defined in “Related Definitions” section for ReadExtended() function above.
**BufferSize**

The size of the **Buffer** in bytes. This must be a multiple of the intrinsic block size of the device.

**Buffer**

A pointer to the source buffer for the data.

**Index**

A pointer to the configuration table index. This is optional.

**CryptoIvInput**

A pointer to a buffer that contains additional cryptographic parameters as required by the capability referenced by the configuration table index, such as cryptographic initialization vector.

**Description**

The **WriteExtended()** function allows the caller to perform a storage device write operation. The function encrypts the requested number of blocks inline if **Index** is specified and then writes them to the device. All the blocks are encrypted (if encryption requested) and written, or an error is returned.

If there is no media in the device, the function returns **EFI_NO_MEDIA**. If the **MediaId** is not the ID for the current media in the device, the function returns **EFI_MEDIA_CHANGED**.

If **EFI_DEVICE_ERROR**, **EFI_NO_MEDIA**, **EFI_WRITE_PROTECTED** or **EFI_MEDIA_CHANGED** is returned and non-blocking I/O is being used, the **Event** associated with this request will not be signaled.

In addition to standard storage transaction parameters (LBA, IO size, and buffer), this command will also specify a configuration table **Index** and a **CryptoIvInput** when data has to be encrypted inline by the controller before being written to the storage device. If no **Index** parameter is specified, no encryption is performed.

**Status Codes Returned**

<table>
<thead>
<tr>
<th>Status Code</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>EFI_SUCCESS</strong></td>
<td>The request to encrypt (optionally) and write was queued if Event is not NULL. The data was encrypted (optionally) and written correctly to the device if the <strong>Event</strong> is NULL.</td>
</tr>
<tr>
<td><strong>EFI_WRITE_PROTECTED</strong></td>
<td>The device cannot be written to.</td>
</tr>
<tr>
<td><strong>EFI_NO_MEDIA</strong></td>
<td>There is no media in the device.</td>
</tr>
<tr>
<td><strong>EFI_MEDIA_CHANGED</strong></td>
<td>The <strong>MediaId</strong> is not for the current media.</td>
</tr>
<tr>
<td><strong>EFI_DEVICE_ERROR</strong></td>
<td>The device reported an error while attempting to encrypt blocks or to perform the write operation.</td>
</tr>
<tr>
<td><strong>EFI_BAD_BUFFER_SIZE</strong></td>
<td>The <strong>BufferSize</strong> parameter is not a multiple of the intrinsic block size of the device.</td>
</tr>
<tr>
<td><strong>EFI_INVALID_PARAMETER</strong></td>
<td><strong>This is NULL</strong>, or the write request contains LBAs that are not valid, or the buffer is not on proper alignment.</td>
</tr>
<tr>
<td><strong>EFI_INVALID_PARAMETER</strong></td>
<td><strong>CryptoIvInput</strong> is incorrect.</td>
</tr>
<tr>
<td><strong>EFI_OUT_OF_RESOURCES</strong></td>
<td>The request could not be completed due to a lack of resources.</td>
</tr>
</tbody>
</table>
**EFI_BLOCK_IO_CRYPTO_PROTOCOL.FlushBlocks()**

**Summary**
Flushes all modified data to a physical block device.

**Prototype**

typedef EFI_STATUS
(EIFIAPI *EFI_BLOCK_IO_CRYPTO_FLUSH) (  
  IN EFI_BLOCK_IO_CRYPTO_PROTOCOL *This,
  IN OUT EFI_BLOCK_IO_CRYPTO_TOKEN *Token
);

**Parameters**

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>This</td>
<td>Pointer to the <strong>EFI_BLOCK_IO_CRYPTO_PROTOCOL</strong> instance.</td>
</tr>
<tr>
<td>Token</td>
<td>A pointer to the token associated with the transaction. Type <strong>EFI_BLOCK_IO_CRYPTO_TOKEN</strong> is defined in “Related Definitions” section for <strong>ReadExtended()</strong> function above.</td>
</tr>
</tbody>
</table>

**Description**
The **FlushBlocks()** function flushes all modified data to the physical block device. Any modified data that has to be encrypted must have been already encrypted as a part of **WriteExtended()** operation – inline crypto operation cannot be a part of flush operation.

All data written to the device prior to the flush must be physically written before returning **EFI_SUCCESS** from this function. This would include any cached data the driver may have cached, and cached data the device may have cached. A flush may cause a read request following the flush to force a device access.

If **EFI_DEVICE_ERROR**, **EFI_NO_MEDIA**, **EFI_WRITE_PROTECTED** or **EFI_MEDIA_CHANGED** is returned and non-blocking I/O is being used, the **Event** associated with this request will not be signaled.

**Status Codes Returned**

<table>
<thead>
<tr>
<th>Code</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>EFI_SUCCESS</td>
<td>The flush request was queued if <strong>Event</strong> is not NULL. All outstanding data was written correctly to the device if the <strong>Event</strong> is NULL.</td>
</tr>
<tr>
<td>EFI_DEVICE_ERROR</td>
<td>The device reported an error while attempting to write data.</td>
</tr>
<tr>
<td>EFI_WRITE_PROTECTED</td>
<td>The device cannot be written to.</td>
</tr>
<tr>
<td>EFI_NO_MEDIA</td>
<td>There is no media in the device.</td>
</tr>
<tr>
<td>EFI_MEDIA_CHANGED</td>
<td>The <strong>MediaId</strong> is not for the current media.</td>
</tr>
<tr>
<td>EFI_INVALID_PARAMETER</td>
<td><strong>This</strong> is <strong>NULL</strong>.</td>
</tr>
<tr>
<td>EFI_OUT_OF_RESOURCES</td>
<td>The request could not be completed due to a lack of resources.</td>
</tr>
</tbody>
</table>
13.12 Erase Block Protocol

**EFI_ERASE_BLOCK_PROTOCOL**

**Summary**
This protocol provides the ability for a device to expose erase functionality. This optional protocol is installed on the same handle as the **EFI_BLOCK_IO_PROTOCOL** or **EFI_BLOCK_IO2_PROTOCOL**.

**GUID**

```c
#define EFI_ERASE_BLOCK_PROTOCOL_GUID \
{0x95A9A93E, 0xA86E, 0x4926, \ 
{0xaa, 0xef, 0x99, 0x18, 0xe7, 0x72, 0xd9, 0x87}}
```

**Revision Number**

```c
#define EFI_ERASE_BLOCK_PROTOCOL_REVISION ((2<<16) | (60))
```

**Protocol Interface Structure**

```c
typedef struct _EFI_ERASE_BLOCK_PROTOCOL {
  UINT64 Revision;
  UINT32 EraseLengthGranularity;
  EFI_BLOCK_ERASE EraseBlocks;
} EFI_ERASE_BLOCK_PROTOCOL;
```

**Parameters**

- **Revision**
  The revision to which the **EFI_ERASE_BLOCK_PROTOCOL** adheres. All future revisions must be backwards compatible. If a future version is not backwards compatible, it is not the same GUID.

- **EraseLengthGranularity**
  Returns the erase length granularity as a number of logical blocks. A value of 1 means the erase granularity is one logical block.

- **EraseBlocks**
  Erase the requested number of blocks from the device. See the **EraseBlocks()** function description.

**EFI_ERASE_BLOCK_PROTOCOL.EraseBlocks()**

**Summary**
Erase a specified number of device blocks.
Prototype

```
Prototype

Typedef
  EFI_STATUS
  (EFIAPI *EFI_BLOCK_ERASE)
  IN EFI_BLOCK_IO_PROTOCOL *This,
  IN UINT32 MediaId,
  IN EFI_LBA LBA,
  IN OUT EFI_ERASE_BLOCK_TOKEN *Token,
  IN UINTN Size);
```

Parameters

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>This</td>
<td>Indicates a pointer to the calling context. Type is defined in the EFI_ERASE_BLOCK_PROTOCOL description.</td>
</tr>
<tr>
<td>MediaId</td>
<td>The media ID that the erase request is for.</td>
</tr>
<tr>
<td>LBA</td>
<td>The starting logical block address to be erased. The caller is responsible for erasing only legitimate locations. Type EFI_LBA is defined in the EFI_BLOCK_IO_PROTOCOL description.</td>
</tr>
<tr>
<td>Token</td>
<td>A pointer to the token associated with the transaction. Type EFI_ERASE_BLOCK_TOKEN is defined in &quot;Related Definitions&quot; below.</td>
</tr>
<tr>
<td>Size</td>
<td>The size in bytes to be erased. This must be a multiple of the physical block size of the device.</td>
</tr>
</tbody>
</table>

Description

The EraseBlocks() function erases the requested number of device blocks. Upon the successful execution of EraseBlocks() with an EFI_SUCCESS return code, any subsequent reads of the same LBA range would return an initialized/formatted value.

If there is no media in the device, the function returns EFI_NO_MEDIA. If the MediaId is not the ID for the current media in the device, the function returns EFI_MEDIA_CHANGED. The function must return EFI_NO_MEDIA or EFI_MEDIA_CHANGED even if LBA or Size are invalid so the caller can probe for changes in media state.

It is the intention of the EraseBlocks() operation to be at least as performant as writing zeroes to each of the specified LBA locations while ensuring the equivalent security.

On some devices, the granularity of the erasable units is defined by EraseLengthGranularity which is the smallest number of consecutive blocks which can be addressed for erase. The size of the EraseLengthGranularity is device specific and can be obtained from EFI_ERASE_BLOCK_MEDIA structure. The fields of EFI_ERASE_MEDIA are not the same as EFI_BLOCK_IO_MEDIA, so look at the EFI_BLOCK_IO_PROTOCOL and/or EFI_BLOCK_IO2_PROTOCOL on the handle for the complete list of fields, if needed. For optimal performance, the starting LBA to be erased shall be EraseLengthGranularity aligned and the Size shall be an integer multiple of an EraseLengthGranularity.
Related Definitions

typedef struct {
  EFI_EVENT    Event;
  EFI_STATUS   TransactionStatus;
} EFI_ERASE_BLOCK_TOKEN;

**Event**
If `Event` is **NULL**, then blocking I/O is performed. If `Event` is not **NULL**, and non-blocking I/O is supported, then non-blocking I/O is performed, and `Event` will be signaled when the erase request is completed.

**TransactionStatus**
Defines whether the signaled event encountered an error.

### Status Codes Returned

<table>
<thead>
<tr>
<th>Code</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>EFI_SUCCESS</td>
<td>The erase request was queued if <code>Event</code> is not <strong>NULL</strong>. The data was erased correctly to the device if the <code>Event</code> is <strong>NULL</strong> to the device.</td>
</tr>
<tr>
<td>EFI_WRITE_PROTECTED</td>
<td>The device cannot be erased due to write protection.</td>
</tr>
<tr>
<td>EFI_DEVICE_ERROR</td>
<td>The device reported an error while attempting to perform the erase operation.</td>
</tr>
<tr>
<td>EFI_INVALID_PARAMETER</td>
<td>The erase request contains LBAs that are not valid.</td>
</tr>
<tr>
<td>EFI_NO_MEDIA</td>
<td>There is no media in the device.</td>
</tr>
<tr>
<td>EFI_MEDIA_CHANGED</td>
<td>The <code>MediaId</code> is not for the current media.</td>
</tr>
</tbody>
</table>

#### 13.13 ATA Pass Thru Protocol

**EFI_ATA_PASS_THRU_PROTOCOL**

This section provides a detailed description of the **EFI_ATA_PASS_THRU_PROTOCOL**.

**Summary**

Provides services that allow ATA commands to be sent to ATA Devices attached to an ATA controller. Packet-based commands would be sent to ATAPI devices only through the Extended SCSI Pass Thru Protocol. While the **ATA_PASS_THRU** interface would expose an interface to the underlying ATA devices on an ATA controller, **EXT_SCSI_PASS_THRU** is responsible for exposing a packet-based command interface for the ATAPI devices on the same ATA controller.
GUID

```csharp
#define EFI_ATA_PASS_THRU_PROTOCOL_GUID
{0x1d3de7f0,0x807,0x424f,
 {0xaa,0x69,0x11,0xa5,0x19,0xe4,0x19,0x6f}}
```

Protocol Interface Structure

```csharp
typedef struct _EFI_ATA_PASS_THRU_PROTOCOL {
    EFI_ATA_PASS_THRU_MODE          *Mode;
    EFI_ATA_PASS_THRU_PASSTHRU      PassThru;
    EFI_ATA_PASS_THRU_GET_NEXT_PORT GetNextPort;
    EFI_ATA_PASS_THRU_GET_NEXT_DEVICE GetNextDevice;
    EFI_ATA_PASS_THRU_BUILD_DEVICE_PATH BuildDevicePath;
    EFI_ATA_PASS_THRU_GET_DEVICE    GetDevice;
    EFI_ATA_PASS_THRU_RESET_PORT    ResetPort;
    EFI_ATA_PASS_THRU_RESET_DEVICE  ResetDevice;
} EFI_ATA_PASS_THRU_PROTOCOL;
```

Parameters

* Mode

A pointer to the ` EFI_ATA_PASS_THRU_MODE ` data for this ATA controller. ` EFI_ATA_PASS_THRU_MODE ` is defined in “Related Definitions” below.

* PassThru

Sends an ATA command to an ATA device that is connected to the ATA controller. See the `PassThru()` function description.

* GetNextPort

Retrieves the list of legal ports for ATA devices on an ATA controller. See the `GetNextPort()` function description.

* GetNextDevice

Retrieves the list of legal ATA devices on a specific port of an ATA controller. See the `GetNextDevice()` function description.

* BuildDevicePath

Allocates and builds a device path node for an ATA Device on an ATA controller. See the `BuildDevicePath()` function description.

* GetDevice

Translates a device path node to a port and port multiplier port. See the `GetDevice()` function description.

* ResetPort

Resets an ATA port or channel (PATA). This operation resets all the ATA devices connected to the ATA port or channel. See the `ResetPort()` function description.
ResetDevice

Resets an ATA device that is connected to the ATA controller. See the ResetDevice() function description.

Note: The following data values in the EFI_ATA_PASS_THRU_MODE interface are read-only.

Attributes

Additional information on the attributes of the ATA controller. See “Related Definitions” below for the list of possible attributes.

IoAlign

Supplies the alignment requirement for any buffer used in a data transfer. IoAlign values of 0 and 1 mean that the buffer can be placed anywhere in memory. Otherwise, IoAlign must be a power of 2, and the requirement is that the start address of a buffer must be evenly divisible by IoAlign with no remainder.

Related Definitions

typedef struct {
    UINT32 Attributes;
    UINT32 IoAlign;
} EFI_ATA_PASS_THRU_MODE;

#define EFI_ATA_PASS_THRU_ATTRIBUTES_PHYSICAL   0x0001
#define EFI_ATA_PASS_THRU_ATTRIBUTES_LOGICAL    0x0002
#define EFI_ATA_PASS_THRU_ATTRIBUTES_NONBLOCKIO 0x0004

EFI_ATA_PASS_THRU_ATTRIBUTES_PHYSICAL

If this bit is set, then the EFI_ATA_PASS_THRU_PROTOCOL interface is for physical devices on the ATA controller.

EFI_ATA_PASS_THRU_ATTRIBUTES_LOGICAL

If this bit is set, then the EFI_ATA_PASS_THRU_PROTOCOL interface is for logical devices on the ATA controller.

EFI_ATA_PASS_THRU_ATTRIBUTES_NONBLOCKIO

If this bit is set, then the EFI_ATA_PASS_THRU_PROTOCOL interface supports non-blocking I/O. Every EFI_ATA_PASS_THRU_PROTOCOL must support blocking I/O. The support of non-blocking I/O is optional.
Description

The **EFI_ATA_PASS_THRU_PROTOCOL** provides information about an ATA controller and the ability to send ATA Command Blocks to any ATA device attached to that ATA controller. To send ATAPI command blocks to ATAPI device attached to that ATA controller, use the **EXT_SCSI_PASS_THRU_PROTOCOL** interface.

The ATAPI devices support a small set of the non-packet-based ATA commands. The **EFI_ATA_PASS_THRU_PROTOCOL** may be used to send such ATA commands to ATAPI devices.

The printable name for the controller can be provided through the **EFI_COMPONENT_NAME2_PROTOCOL** for multiple languages.

The **Attributes** field of the Mode member of the **EFI_ATA_PASS_THRU_PROTOCOL** interface tells if the interface is for physical ATA devices or logical ATA devices. Drivers for non-RAID ATA controllers will set both the **EFI_ATA_PASS_THRU_ATTRIBUTES_PHYSICAL**, and the **EFI_ATA_PASS_THRU_ATTRIBUTES_LOGICAL** bits.

Drivers for RAID controllers that allow access to the physical devices and logical devices will produce two **EFI_ATA_PASS_THRU_PROTOCOL** interfaces: one with the just the **EFI_ATA_PASS_THRU_ATTRIBUTES_PHYSICAL** bit set and another with just the **EFI_ATA_PASS_THRU_ATTRIBUTES_LOGICAL** bit set. One interface can be used to access the physical devices attached to the RAID controller, and the other can be used to access the logical devices attached to the RAID controller for its current configuration.

Drivers for RAID controllers that do not allow access to the physical devices will produce one **EFI_ATA_PASS_THROUGH_PROTOCOL** interface with just the **EFI_ATA_PASS_THRU_ATTRIBUTES_LOGICAL** bit set. The interface for logical devices can also be used by a file system driver to mount the RAID volumes. An **EFI_ATA_PASS_THRU_PROTOCOL** with neither **EFI_ATA_PASS_THRU_ATTRIBUTES_LOGICAL** nor **EFI_ATA_PASS_THRU_ATTRIBUTES_PHYSICAL** set is an illegal configuration.

The **Attributes** field also contains the **EFI_ATA_PASS_THRU_ATTRIBUTES_NONBLOCKIO** bit. All **EFI_ATA_PASS_THRU_PROTOCOL** interfaces must support blocking I/O. If this bit is set, then the interface supports both blocking I/O and non-blocking I/O.

Each **EFI_ATA_PASS_THRU_PROTOCOL** instance must have an associated device path. Typically this will have an ACPI device path node and a PCI device path node, although variation will exist.

Additional information about the ATA controller can be obtained from protocols attached to the same handle as the **EFI_ATA_PASS_THRU_PROTOCOL**, or one of its parent handles. This would include the device I/O abstraction used to access the internal registers and functions of the ATA controller.
This protocol may also be used for PATA devices (or devices in a PATA-compatible mode). PATA devices are mapped to ports and port multiplier ports using the following table:

<table>
<thead>
<tr>
<th>PATA Device Connection</th>
<th>Emulated Port Number</th>
<th>Emulated Port Multiplier Port Number</th>
</tr>
</thead>
<tbody>
<tr>
<td>Primary Master</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>Primary Slave</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>Secondary Master</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>Secondary Slave</td>
<td>1</td>
<td>1</td>
</tr>
</tbody>
</table>

**EFI_ATA_PASS_THRU_PROTOCOL.PassThru()**

**Summary**

Sends an ATA command to an ATA device that is attached to the ATA controller. This function supports both blocking I/O and non-blocking I/O. The blocking I/O functionality is required, and the non-blocking I/O functionality is optional.

**Prototype**

```c
typedef EFI_STATUS (EFIAPI *EFI_ATA_PASS_THRU_PASSTHRU) (  
    IN   EFI_ATA_PASS_THRU_PROTOCOL *This,  
    IN   UINT16 Port,  
    IN   UINT16 PortMultiplierPort,  
    IN OUT EFI_ATA_PASS_THRU_COMMAND_PACKET *Packet,  
    IN   EFI_EVENT Event OPTIONAL  
);
```

**Parameters**

*This*  
A pointer to the **EFI_ATA_PASS_THRU_PROTOCOL** instance.

*Port*  
The port number of the ATA device to send the command.

*PortMultiplierPort*  
The port multiplier port number of the ATA device to send the command. If there is no port multiplier, then specify 0xFFFF.

*Packet*  
A pointer to the ATA command to send to the ATA device specified by *Port* and *PortMultiplierPort*. See “Related Definitions” below for a description of **EFI_ATA_PASS_THRU_COMMAND_PACKET**.
Event

If non-blocking I/O is not supported then Event is ignored, and blocking I/O is performed. If Event is **NULL**, then blocking I/O is performed. If Event is not **NULL** and non blocking I/O is supported, then non-blocking I/O is performed, and Event will be signaled when the ATA command completes.

Related Definitions

```c
typedef struct {
  EFI_ATA_STATUS_BLOCK *Asb;
  EFI_ATA_COMMAND_BLOCK *Acb;
  UINT64 Timeout;
  VOID *InDataBuffer;
  VOID *OutDataBuffer;
  UINT32 InTransferLength;
  UINT32 OutTransferLength;
  EFI_ATA_PASS_THRU_CMD_PROTOCOL Protocol;
  EFI_ATA_PASS_THRU_LENGTH Length;
} EFI_ATA_PASS_THRU_COMMAND_PACKET;
```

Timeout

The timeout, in 100 ns units, to use for the execution of this ATA command. A Timeout value of 0 means that this function will wait indefinitely for the ATA command to execute. If Timeout is greater than zero, then this function will return **EFI_TIMEOUT** if the time required to execute the ATA command is greater than Timeout.

InDataBuffer

A pointer to the data buffer to transfer between the ATA controller and the ATA device for read and bidirectional commands. For all write and non data commands where InTransferLength is 0 this field is optional and may be **NULL**. If this field is not **NULL**, then it must be aligned on the boundary specified by the IoAlign field in the **EFI_ATA_PASS_THRU_MODE** structure.

OutDataBuffer

A pointer to the data buffer to transfer between the ATA controller and the ATA device for write or bidirectional commands. For all read and non data commands where OutTransferLength is 0 this field is optional and may be **NULL**. If this field is not **NULL**, then it must be aligned on the boundary specified by the IoAlign field in the **EFI_ATA_PASS_THRU_MODE** structure.

InTransferLength

On input, the size, in bytes, of InDataBuffer. On output, the number of bytes transferred between the ATA controller and the ATA device. If InTransferLength is larger than the ATA controller can handle, no data will be transferred, InTransferLength will be updated to contain the number of bytes that the ATA controller is able to transfer, and **EFI_BAD_BUFFER_SIZE** will be returned.
**OutTransferLength**

On Input, the size, in bytes of OutDataBuffer. On Output, the Number of bytes transferred between ATA Controller and the ATA device. If OutTransferLength is larger than the ATA controller can handle, no data will be transferred, OutTransferLength will be updated to contain the number of bytes that the ATA controller is able to transfer, and EFI_BAD_BUFFER_SIZE will be returned.

**Asb**

A pointer to the sense data that was generated by the execution of the ATA command. It must be aligned to the boundary specified in the IoAlign field in the EFI_ATA_PASS_THRU_MODE structure.

**Acb**

A pointer to buffer that contains the Command Data Block to send to the ATA device specified by Port and PortMultiplierPort.

**Protocol**

Specifies the protocol used when the ATA device executes the command. Type EFI_ATA_PASS_THRU_CMD_PROTOCOL is defined below.

**Length**

Specifies the way in which the ATA command length is encoded. Type EFI_ATA_PASS_THRU_LENGTH is defined below.
typedef struct _EFI_ATA_COMMAND_BLOCK {
    UINT8 Reserved1[2];
    UINT8 AtaCommand;
    UINT8 AtaFeatures;
    UINT8 AtaSectorNumber;
    UINT8 AtaCylinderLow;
    UINT8 AtaCylinderHigh;
    UINT8 AtaDeviceHead;
    UINT8 AtaSectorNumberExp;
    UINT8 AtaCylinderLowExp;
    UINT8 AtaCylinderHighExp;
    UINT8 AtaFeaturesExp;
    UINT8 AtaSectorCount;
    UINT8 AtaSectorCountExp;
    UINT8 Reserved2[6];
} EFI_ATA_COMMAND_BLOCK;

typedef struct _EFI_ATA_STATUS_BLOCK {
    UINT8 Reserved1[2];
    UINT8 AtaStatus;
    UINT8 AtaError;
    UINT8 AtaSectorNumber;
    UINT8 AtaCylinderLow;
    UINT8 AtaCylinderHigh;
    UINT8 AtaDeviceHead;
    UINT8 AtaSectorNumberExp;
    UINT8 AtaCylinderLowExp;
    UINT8 AtaCylinderHighExp;
    UINT8 Reserved2;
    UINT8 AtaSectorCount;
    UINT8 AtaSectorCountExp;
    UINT8 Reserved3[6];
} EFI_ATA_STATUS_BLOCK;

typedef UINT8 EFI_ATA_PASS_THRU_CMD_PROTOCOL;

#define EFI_ATA_PASS_THRU_PROTOCOL_ATA_HARDWARE_RESET 0x00
#define EFI_ATA_PASS_THRU_PROTOCOL_ATA_SOFTWARE_RESET 0x01
#define EFI_ATA_PASS_THRU_PROTOCOL_ATA_NON_DATA 0x02
#define EFI_ATA_PASS_THRU_PROTOCOLPIO_DATA_IN 0x04
#define EFI_ATA_PASS_THRU_PROTOCOLPIO_DATA_OUT 0x05
#define EFI_ATA_PASS_THRU_PROTOCOL_DMA 0x06
#define EFI_ATA_PASS_THRU_PROTOCOL_DMA_QUEUED 0x07
#define EFI_ATA_PASS_THRU_PROTOCOL_DEVICE_DIAGNOSTIC 0x08
#define EFI_ATA_PASS_THRU_PROTOCOL_DEVICE_RESET 0x09
typedef UINT8 EFI_ATA_PASS_THRU_LENGTH;

#define EFI_ATA_PASS_THRU_LENGTH_BYTES 0x80

#define EFI_ATA_PASS_THRU_LENGTH_MASK 0x70
#define EFI_ATA_PASS_THRU_LENGTH_NO_DATA_TRANSFER 0x00
#define EFI_ATA_PASS_THRU_LENGTH_FEATURES 0x10
#define EFI_ATA_PASS_THRU_LENGTH_SECTOR_COUNT 0x20
#define EFI_ATA_PASS_THRU_LENGTH_TPSIU 0x30
#define EFI_ATA_PASS_THRU_LENGTH_COUNT 0x0F

Description

The PassThru() function sends the ATA command specified by Packet to the ATA device specified by Port and PortMultiplierPort. If the driver supports non-blocking I/O and Event is not NULL, then the driver will return immediately after the command is sent to the selected device, and will later signal Event when the command has completed.

If the driver supports non-blocking I/O and Event is NULL, then the driver will send the command to the selected device and block until it is complete. If the driver does not support non-blocking I/O, then the Event parameter is ignored, and the driver will send the command to the selected device and block until it is complete.

If Packet is successfully sent to the ATA device, then EFI_SUCCESS is returned. If Packet cannot be sent because there are too many packets already queued up, then EFI_NOT_READY is returned. The caller may retry Packet at a later time. If a device error occurs while sending the Packet, then EFI_DEVICE_ERROR is returned. If a timeout occurs during the execution of Packet, then EFI_TIMEOUT is returned.

If Port or PortMultiplierPort are not in a valid range for the ATA controller, then EFI_INVALID_PARAMETER is returned. If InDataBuffer, OutDataBuffer or Asb do not meet the alignment requirement specified by the IoAlign field of the EFI_ATA_PASS_THRU_MODE structure, then EFI_INVALID_PARAMETER is returned. If any of the other fields of Packet are invalid, then EFI_INVALID_PARAMETER is returned.

If the data buffer described by InDataBuffer and InTransferLength is too big to be transferred in a single command, then no data is transferred and EFI_BAD_BUFFER_SIZE is returned. The number of bytes that can be transferred in a single command are returned in InTransferLength. If the data buffer described by OutDataBuffer and OutTransferLength is too big to be transferred in a single command, then no data is transferred and EFI_BAD_BUFFER_SIZE is returned. The number of bytes that can be transferred in a single command are returned in OutTransferLength.
If the command described in Packet is not supported by the host adapter, then EFI_UNSUPPORTED is returned.

If EFI_SUCCESS, EFI_BAD_BUFFER_SIZE, EFI_DEVICE_ERROR, or EFI_TIMEOUT is returned, then the caller must examine Asb.

If non-blocking I/O is being used, then the status fields in Packet will not be valid until the Event associated with Packet is signaled.

If EFI_NOT_READY, EFI_INVALID_PARAMETER or EFI_UNSUPPORTED is returned, then Packet was never sent, so the status fields in Packet are not valid. If non-blocking I/O is being used, the Event associated with Packet will not be signaled.

This function will determine if data transfer is necessary based on the Acb->Protocol and Acb->Length fields. The Acb->AtaCommand field is ignored except to copy it into the ATA Command register. The following table describes special programming considerations based on the protocol specified by Acb->Protocol.

Table 13-3 Special programming considerations

<table>
<thead>
<tr>
<th>Protocol Value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>EFI_ATA_PASS_THRU_PROTOCOL_ATA_HARDWARE_RESET</td>
<td>For PATA devices, then RST- is asserted. For SATA devices, then COMRESET will be issued.</td>
</tr>
<tr>
<td>EFI_ATA_PASS_THRU_PROTOCOL_ATA_SOFTWARE_RESET</td>
<td>A software reset will be issued to the ATA device.</td>
</tr>
<tr>
<td>EFI_ATA_PASS_THRU_PROTOCOLPIO_DATA_IN - EFI_ATA_PASS_THRU_PROTOCOL_FPDMA</td>
<td>The command is sent to the ATA device. If the value is inappropriate for the command specified by Acb-&gt;AtaCommand, the results are undefined.</td>
</tr>
<tr>
<td>EFI_ATA_PASS_THRU_RETURN_RESPONSE</td>
<td>This command will only return the contents of the ATA status block.</td>
</tr>
</tbody>
</table>

The ATA host and the ATA device should already be configured for the PIO, DMA, and UDMA transfer rates that are supported by the ATA controller and the ATA device. The results of changing the device’s timings using this function are undefined.

If Packet->Length is not set to EFI_ATA_PASS_THRU_LENGTH_NO_DATA_TRANSFER, then if EFI_ATA_PASS_THRU_LENGTH_BYTES is set in Packet->Length, then Packet->InTransferLength and Packet->OutTransferLength are interpreted as bytes.

If Packet->Length is not set to EFI_ATA_PASS_THRU_LENGTH_NO_DATA_TRANSFER, then if EFI_ATA_PASS_THRU_LENGTH_BYTES is clear in Packet->Length, then Packet->InTransferLength and Packet->OutTransferLength are interpreted as blocks.

If Packet->Length is set to EFI_ATA_PASS_THRU_LENGTH_SECTOR_COUNT, then the transfer length will be programmed into Acb->AtaSectorCount.
If `Packet->Length` is set to `EFI_ATA_PASS_THRU_LENGTH_TPSIU`, then the transfer length will be programmed into the TPSIU.

- For PIO data transfers, the number of sectors to transfer is $2 \times (Packet->Length \& EFI_ATA_PASS_THRU_LENGTH_COUNT)$.

For all commands, the contents of the ATA status block will be returned in `Asb`.

### Status Codes Returned

<table>
<thead>
<tr>
<th>Code</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>EFI_SUCCESS</code></td>
<td>The ATA command was sent by the host. For bi-directional commands, <code>InTransferLength</code> bytes were transferred from <code>InDataBuffer</code>. For write and bi-directional commands, <code>OutTransferLength</code> bytes were transferred by <code>OutDataBuffer</code>. See <code>Asb</code> for additional status information.</td>
</tr>
<tr>
<td><code>EFI_BAD_BUFFER_SIZE</code></td>
<td>The ATA command was not executed. The number of bytes that could be transferred is returned in <code>InTransferLength</code>. For write and bi-directional commands, <code>OutTransferLength</code> bytes were transferred by <code>OutDataBuffer</code>. See <code>Asb</code> for additional status information.</td>
</tr>
<tr>
<td><code>EFI_NOT_READY</code></td>
<td>The ATA command could not be sent because there are too many ATA commands already queued. The caller may retry again later.</td>
</tr>
<tr>
<td><code>EFI_DEVICE_ERROR</code></td>
<td>A device error occurred while attempting to send the ATA command. See <code>Asb</code> for additional status information.</td>
</tr>
<tr>
<td><code>EFI_INVALID_PARAMETER</code></td>
<td><code>Port</code>, <code>PortMultiplierPort</code>, or the contents of <code>Acb</code> are invalid. The ATA command was not sent, so no additional status information is available.</td>
</tr>
<tr>
<td><code>EFI_UNSUPPORTED</code></td>
<td>The command described by the ATA command is not supported by the host adapter. The ATA command was not sent, so no additional status information is available.</td>
</tr>
<tr>
<td><code>EFI_TIMEOUT</code></td>
<td>A timeout occurred while waiting for the ATA command to execute. See <code>Asb</code> for additional status information.</td>
</tr>
</tbody>
</table>

### `EFI_ATA_PASS_THRU_PROTOCOL.GetNextPort()`

**Summary**

Used to retrieve the list of legal port numbers for ATA devices on an ATA controller. These can either be the list of ports where ATA devices are actually present or the list of legal port numbers for the ATA controller. Regardless, the caller of this function must probe the port number returned to see if an ATA device is actually present at that location on the ATA controller.
Prototype

```c
typedef EFI_STATUS
    (EFIAPIC *EFI_ATA_PASS_THRU_GET_NEXT_PORT) ( 
    IN EFI_ATA_PASS_THRU_PROTOCOL *This,
    IN OUT UINT16 *Port
    );
```

Parameters

- **This**
  A pointer to the `EFI_ATA_PASS_THRU_PROTOCOL` instance.

- **Port**
  On input, a pointer to the port number on the ATA controller. On output, a pointer to the next port number on the ATA controller. An input value of `0xFFFF` retrieves the first port number on the ATA controller.

Description

The `GetNextPort()` function retrieves the port number on an ATA controller. If on input `Port` is `0xFFFF`, then the port number of the first port on the ATA controller is returned in `Port` and `EFI_SUCCESS` is returned.

If `Port` is the port number that was returned on the previous call to `GetNextPort()`, then the port number of the next port on the ATA controller is returned in `Port`, and `EFI_SUCCESS` is returned.

If `Port` is not `0xFFFF` and `Port` was not returned on the previous call to `GetNextPort()`, then `EFI_INVALID_PARAMETER` is returned.

If `Port` is the port number of the last port on the ATA controller, then `EFI_NOT_FOUND` is returned.

Status Codes Returned

<table>
<thead>
<tr>
<th>Status Code</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>EFI_SUCCESS</td>
<td>The next port number on the ATA controller was returned in <code>Port</code>.</td>
</tr>
<tr>
<td>EFI_NOT_FOUND</td>
<td>There are no more ports on this ATA controller.</td>
</tr>
</tbody>
</table>
| EFI_INVALID_PARAMETER | `Port` is not `0xFFFF` and `Port` was not returned on a previous call to `GetNextPort()`.

`EFI_ATA_PASS_THRU_PROTOCOL.GetNextDevice()`

Summary

Used to retrieve the list of legal port multiplier port numbers for ATA devices on a port of an ATA controller. These can either be the list of port multiplier ports where ATA devices are actually present on port or the list of legal port multiplier ports on that port. Regardless, the caller of this function must probe the port number and port multiplier port number returned to see if an ATA device is actually present.
Prototype

```
typedef EFI_STATUS
  (EFIAPI *EFI_ATA_PASS_THRU_GET_NEXT_DEVICE) (
   IN EFI_ATA_PASS_THRU_PROTOCOL *This,
   IN UINT16 Port,
   IN OUT UINT16 *PortMultiplierPort
  );
```

Parameters

- **This**: A pointer to the `EFI_ATA_PASS_THRU_PROTOCOL` instance.
- **Port**: The port number present on the ATA controller.
- **PortMultiplierPort**: On input, a pointer to the port multiplier port number of an ATA device present on the ATA controller. If on input a `PortMultiplierPort` of `0xFFFF` is specified, then the port multiplier port number of the first ATA device is returned. On output, a pointer to the port multiplier port number of the next ATA device present on an ATA controller.

Description

The `GetNextDevice()` function retrieves the port multiplier port number of an ATA device present on a port of an ATA controller.

- If `PortMultiplierPort` points to a port multiplier port number value that was returned on a previous call to `GetNextDevice()`, then the port multiplier port number of the next ATA device on the port of the ATA controller is returned in `PortMultiplierPort`, and `EFI_SUCCESS` is returned.
- If `PortMultiplierPort` points to `0xFFFF`, then the port multiplier port number of the first ATA device on port of the ATA controller is returned in `PortMultiplierPort` and `EFI_SUCCESS` is returned.
- If `PortMultiplierPort` is not `0xFFFF` and the value pointed to by `PortMultiplierPort` was not returned on a previous call to `GetNextDevice()`, then `EFI_INVALID_PARAMETER` is returned.
- If `PortMultiplierPort` is the port multiplier port number of the last ATA device on the port of the ATA controller, then `EFI_NOT_FOUND` is returned.

Status Codes Returned

<table>
<thead>
<tr>
<th>Code</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>EFI_SUCCESS</td>
<td>The port multiplier port number of the next ATA device on the port of the ATA controller was returned in <code>PortMultiplierPort</code>.</td>
</tr>
<tr>
<td>EFI_NOT_FOUND</td>
<td>There are no more ATA devices on this port of the ATA controller.</td>
</tr>
<tr>
<td>EFI_INVALID_PARAMETER</td>
<td><code>PortMultiplierPort</code> is not <code>0xFFFF</code>, and <code>PortMultiplierPort</code> was not returned on a previous call to <code>GetNextDevice()</code>.</td>
</tr>
</tbody>
</table>
**EFI_ATA_PASS_THRU_PROTOCOL.BuildDevicePath()**

**Summary**
Used to allocate and build a device path node for an ATA device on an ATA controller.

**Prototype**
```
typedef
   EFI_STATUS
   (EFIAPI *EFI_ATA_PASS_THRU_BUILD_DEVICE_PATH) (   
   IN   EFI_ATA_PASS_THRU_PROTOCOL  *This,
   IN   UINT16                     Port,    
   IN   UINT16                     PortMultiplierPort,
   OUT  EFI_DEVICE_PATH_PROTOCOL     **DevicePath
   );
```

**Parameters**
- **This**
  A pointer to the EFI_ATA_PASS_THRU_PROTOCOL instance.
- **Port**
  Port specifies the port number of the ATA device for which a device path node is to be allocated and built.
- **PortMultiplierPort**
  The port multiplier port number of the ATA device for which a device path node is to be allocated and built. If there is no port multiplier, then specify 0xFFFF.
- **DevicePath**
  A pointer to a single device path node that describes the ATA device specified by Port and PortMultiplierPort. This function is responsible for allocating the buffer DevicePath with the boot service AllocatePool(). It is the caller’s responsibility to free DevicePath when the caller is finished with DevicePath.

**Description**
The BuildDevicePath() function allocates and builds a single device node for the ATA device specified by Port and PortMultiplierPort. If the ATA device specified by Port and PortMultiplierPort is not present on the ATA controller, then EFI_NOT_FOUND is returned. If DevicePath is NULL, then EFI_INVALID_PARAMETER is returned. If there are not enough resources to allocate the device path node, then EFI_OUT_OF_RESOURCES is returned.

Otherwise, DevicePath is allocated with the boot service AllocatePool(), the contents of DevicePath are initialized to describe the ATA device specified by Port and PortMultiplierPort, and EFI_SUCCESS is returned.
Status Codes Returned

<table>
<thead>
<tr>
<th>Code</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>EFI_SUCCESS</td>
<td>The device path node that describes the ATA device specified by Port and PortMultiplierPort was allocated and returned in DevicePath.</td>
</tr>
<tr>
<td>EFI_NOT_FOUND</td>
<td>The ATA device specified by Port and PortMultiplierPort does not exist on the ATA controller.</td>
</tr>
<tr>
<td>EFI_INVALID_PARAMETER</td>
<td>DevicePath is NULL.</td>
</tr>
<tr>
<td>EFI_OUT_OF_RESOURCES</td>
<td>There are not enough resources to allocate DevicePath.</td>
</tr>
</tbody>
</table>

EFI_ATA_PASS_THRU_PROTOCOL.GetDevice()

Summary

Used to translate a device path node to a port number and port multiplier port number.

Prototype

```c
typedef EFI_STATUS (EFIAPI *EFI_ATA_PASS_THRU_GET_DEVICE) (
    IN EFI_ATA_PASS_THRU_PROTOCOL *This,
    IN EFI_DEVICE_PATH_PROTOCOL   *DevicePath,
    OUT UINT16                    *Port,
    OUT UINT16                    *PortMultiplierPort
);
```

Parameters

- **This**
  
  A pointer to the EFI_ATA_PASS_THRU_PROTOCOL instance.

- **DevicePath**
  
  A pointer to the device path node that describes an ATA device on the ATA controller.

- **Port**
  
  On return, points to the port number of an ATA device on the ATA controller.

- **PortMultiplierPort**
  
  On return, points to the port multiplier port number of an ATA device on the ATA controller.

Description

The GetDevice() function determines the port and port multiplier port number associated with the ATA device described by DevicePath. If DevicePath is a device path node type that the ATA Pass Thru driver supports, then the ATA Pass Thru driver will attempt to translate the contents DevicePath into a port number and port multiplier port number.

If this translation is successful, then that port number and port multiplier port number are returned in Port and PortMultiplierPort, and EFI_SUCCESS is returned.
If `DevicePath`, `Port`, or `PortMultiplierPort` are `NULL`, then `EFI_INVALID_PARAMETER` is returned.

If `DevicePath` is not a device path node type that the ATA Pass Thru driver supports, then `EFI_UNSUPPORTED` is returned.

If `DevicePath` is a device path node type that the ATA Pass Thru driver supports, but there is not a valid translation from `DevicePath` to a port number and port multiplier port number, then `EFI_NOT_FOUND` is returned.

### Status Codes Returned

<table>
<thead>
<tr>
<th>Code</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>EFI_SUCCESS</td>
<td><code>DevicePath</code> was successfully translated to a port number and port multiplier port number, and they were returned in <code>Port</code> and <code>PortMultiplierPort</code>.</td>
</tr>
<tr>
<td>EFI_INVALID_PARAMETER</td>
<td><code>DevicePath</code> is <code>NULL</code>.</td>
</tr>
<tr>
<td>EFI_INVALID_PARAMETER</td>
<td><code>Port</code> is <code>NULL</code>.</td>
</tr>
<tr>
<td>EFI_INVALID_PARAMETER</td>
<td><code>PortMultiplierPort</code> is <code>NULL</code>.</td>
</tr>
<tr>
<td>EFI_UNSUPPORTED</td>
<td>This driver does not support the device path node type in <code>DevicePath</code>.</td>
</tr>
<tr>
<td>EFI_NOT_FOUND</td>
<td>A valid translation from <code>DevicePath</code> to a port number and port multiplier port number does not exist.</td>
</tr>
</tbody>
</table>

### EFI_ATA_PASS_THRU_PROTOCOL.ResetPort()

**Summary**

Resets a specific port on the ATA controller. This operation also resets all the ATA devices connected to the port.

**Prototype**

```c
typedef EFI_STATUS (EFIAPI *EFI_ATA_PASS_THRU_RESET_PORT) (  
    IN EFI_ATA_PASS_THRU_PROTOCOL *This,  
    IN UINT16 Port  
);
```

**Parameters**

- **This**
  
  A pointer to the `EFI_ATA_PASS_THRU_PROTOCOL` instance.

- **Port**
  
  The port number on the ATA controller.
Description
The **ResetChannel()** function resets an a specific port on an ATA controller. This operation resets all the ATA devices connected to that port. If this ATA controller does not support a reset port operation, then **EFI_UNSUPPORTED** is returned.

If a device error occurs while executing that port reset operation, then **EFI_DEVICE_ERROR** is returned.

If a timeout occurs during the execution of the port reset operation, then **EFI_TIMEOUT** is returned. If the port reset operation is completed, then **EFI_SUCCESS** is returned.

### Status Codes Returned

<table>
<thead>
<tr>
<th>Status Code</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>EFI_SUCCESS</strong></td>
<td>The ATA controller port was reset.</td>
</tr>
<tr>
<td><strong>EFI_UNSUPPORTED</strong></td>
<td>The ATA controller does not support a port reset operation.</td>
</tr>
<tr>
<td><strong>EFI_DEVICE_ERROR</strong></td>
<td>A device error occurred while attempting to reset the ATA port.</td>
</tr>
<tr>
<td><strong>EFI_TIMEOUT</strong></td>
<td>A timeout occurred while attempting to reset the ATA port.</td>
</tr>
</tbody>
</table>

**EFI_ATA_PASS_THRU_PROTOCOL.ResetDevice()**

**Summary**
Resets an ATA device that is connected to an ATA controller.

**Prototype**
```c
typedef
EFI_STATUS
(EFIAPI *EFI_ATA_PASS_THRU_RESET_DEVICE) (  
    IN EFI_ATA_PASS_THRU_PROTOCOL  *This,  
    IN UINT16 Port,  
    IN UINT16 PortMultiplierPort
);
```

**Parameters**
- **This**
  A pointer to the **EFI_ATA_PASS_THRU_PROTOCOL** instance.
- **Port**
  Port represents the port number of the ATA device to be reset.
- **PortMultiplierPort**
  The port multiplier port number of the ATA device to reset. If there is no port multiplier, then specify 0xFFFF.

**Description**
The **ResetDevice()** function resets the ATA device specified by **Port** and **PortMultiplierPort**. If this ATA controller does not support a device reset operation, then **EFI_UNSUPPORTED** is returned.

If **Port** or **PortMultiplierPort** are not in a valid range for this ATA controller, then **EFI_INVALID_PARAMETER** is returned.
If a device error occurs while executing that device reset operation, then `EFI_DEVICE_ERROR` is returned.

If a timeout occurs during the execution of the device reset operation, then `EFI_TIMEOUT` is returned.

If the device reset operation is completed, then `EFI_SUCCESS` is returned.

### Status Codes Returned

<table>
<thead>
<tr>
<th>Status Code</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>EFI_SUCCESS</td>
<td>The ATA device specified by <code>Port</code> and <code>PortMultiplierPort</code> was reset</td>
</tr>
<tr>
<td>EFI_UNSUPPORTED</td>
<td>The ATA controller does not support a device reset operation.</td>
</tr>
<tr>
<td>EFI_INVALID_PARAMETER</td>
<td><code>Port</code> or <code>PortMultiplierPort</code> are invalid.</td>
</tr>
<tr>
<td>EFI_DEVICE_ERROR</td>
<td>A device error occurred while attempting to reset the ATA device specified by <code>Port</code> and <code>PortMultiplierPort</code>.</td>
</tr>
<tr>
<td>EFI_TIMEOUT</td>
<td>A timeout occurred while attempting to reset the ATA device specified by <code>Port</code> and <code>PortMultiplierPort</code>.</td>
</tr>
</tbody>
</table>

### 13.14 Storage Security Command Protocol

This section defines the storage security command protocol. This protocol is used to abstract mass storage devices to allow code running in the EFI boot services environment to send security protocol commands to mass storage devices without specific knowledge of the type of device or controller that manages the device. Functions are defined to send or retrieve security protocol defined data to and from mass storage devices. This protocol shall be supported on all physical and logical storage devices supporting the `EFI_BLOCK_IO_PROTOCOL` or `EFI_BLOCK_IO2_PROTOCOL` in the EFI boot services environment and one of the following command sets (or their alternative) at the bus level:

- TRUSTED SEND/RECEIVE commands of the ATA8-ACS command set or its successor
- SECURITY PROTOCOL IN/OUT commands of the SPC-4 command set or its successor.

If the mass storage device is part of a RAID set, the specific physical device may not support the block IO protocols directly, but they are supported by the logical device defining the RAID set. In this case the `MediaId` parameter may not be available and its value is undefined for this interface.
**EFI_STORAGE_SECURITY_COMMAND_PROTOCOL**

**Summary**
This protocol provides ability to send security protocol commands to mass storage devices.

**GUID**
```
#define EFI_STORAGE_SECURITY_COMMAND_PROTOCOL_GUID \
{0xc88b0b6d, 0x0dfc, 0x49a7,\ 
 {0x9c, 0xb4, 0x49, 0x7, 0x4b, 0x4c, 0x3a, 0x78}}
```

**Protocol Interface Structure**
```c
typedef struct _EFI_STORAGE_SECURITY_COMMAND_PROTOCOL {
    EFI_STORAGE_SECURITY_RECEIVE_DATA ReceiveData;
    EFI_STORAGE_SECURITY_SEND_DATA SendData;
} EFI_STORAGE_SECURITY_COMMAND_PROTOCOL;
```

**Parameters**
- **ReceiveData**
  Issues a security protocol command to the requested device that receives data and/or the result of one or more commands sent by **SendData**. See the **ReceiveData()** function description.

- **SendData**
  Issues a security protocol command to the requested device. See the **SendData()** function description.

**Description**
The **EFI_STORAGE_SECURITY_COMMAND_PROTOCOL** is used to send security protocol commands to a mass storage device. Two types of security protocol commands are supported. **SendData** sends a command with data to a device. **ReceiveData** sends a command that receives data and/or the result of one or more commands sent by **SendData**.

The security protocol command formats supported shall be based on the definition of the SECURITY PROTOCOL IN and SECURITY PROTOCOL OUT commands defined in SPC-4. If the device uses the SCSI command set, no translation is needed in the firmware and the firmware can package the parameters into a SECURITY PROTOCOL IN or SECURITY PROTOCOL OUT command and send the command to the device. If the device uses a non-SCSI command set, the firmware shall map the command and data payload to the corresponding command and payload format defined in the non-SCSI command set (for example, TRUSTED RECEIVE and TRUSTED SEND in ATA8-ACS).

The firmware shall automatically add an **EFI_STORAGE_SECURITY_COMMAND_PROTOCOL** for any storage devices detected during system boot that support SPC-4, ATA8-ACS or their successors.
**EFI_STORAGE_SECURITY_COMMAND_PROTOCOL.ReceiveData()**

**Summary**
Send a security protocol command to a device that receives data and/or the result of one or more commands sent by `SendData`.

**Prototype**
```c
typedef EFI_STATUS
  (EFI_API *EFI_STORAGE_SECURITY_RECEIVE_DATA) (
   IN EFI_STORAGE_SECURITY_COMMAND_PROTOCOL *
   This,
   IN UINT32 MediaId,
   IN UINT64 Timeout,
   IN UINT8 SecurityProtocol,
   IN UINT16 SecurityProtocolSpecificData,
   IN UINTN PayloadBufferSize,
   OUT VOID       *PayloadBuffer,
   OUT UINTN  *PayloadTransferSize
  );
```

**Parameters**
- **This**
  Indicates a pointer to the calling context. Type `EFI_STORAGE_SECURITY_COMMAND_PROTOCOL` is defined in the `EFI_STORAGE_SECURITY_COMMAND_PROTOCOL` description.
- **MediaId**
  ID of the medium to receive data from. If there is no block IO protocol supported by the physical device, the value of MediaId is undefined.
- **Timeout**
  The timeout, in 100ns units, to use for the execution of the security protocol command. A `Timeout` value of 0 means that this function will wait indefinitely for the security protocol command to execute. If `Timeout` is greater than zero, then this function will return `EFI_TIMEOUT` if the time required to execute the receive data command is greater than `Timeout`.
- **SecurityProtocolId**
  The value of the “Security Protocol” parameter of the security protocol command to be sent.
- **SecurityProtocolSpecificData**
  The value of the “Security Protocol Specific” parameter of the security protocol command to be sent. This value is in big-endian format.
- **PayloadBufferSize**
  Size in bytes of the payload data buffer.
- **PayloadBuffer**
  A pointer to a destination buffer to store the security protocol command specific payload data for the security protocol command.
The caller is responsible for having either implicit or explicit ownership of the buffer.

**PayloadTransferSize**

A pointer to a buffer to store the size in bytes of the data written to the payload data buffer.

**Description**

The **ReceiveData** function sends a security protocol command to the given **MediaId**. The security protocol command sent is defined by **SecurityProtocolId** and contains the security protocol specific data **SecurityProtocolSpecificData**. The function returns the data from the security protocol command in **PayloadBuffer**.

For devices supporting the SCSI command set, the security protocol command is sent using the SECURITY PROTOCOL IN command defined in SPC-4.

For devices supporting the ATA command set, the security protocol command is sent using one of the TRUSTED RECEIVE commands defined in ATA8-ACS if **PayloadBufferSize** is non-zero. If the **PayloadBufferSize** is zero, the security protocol command is sent using the Trusted Non-Data command defined in ATA8-ACS.

If **PayloadBufferSize** is too small to store the available data from the security protocol command, the function shall copy **PayloadBufferSize** bytes into the **PayloadBuffer** and return **EFI_WARN_BUFFER_TOO_SMALL**.

If **PayloadBuffer** or **PayloadTransferSize** is **NULL** and **PayloadBufferSize** is non-zero, the function shall return **EFI_INVALID_PARAMETER**.

If the given **MediaId** does not support security protocol commands, the function shall return **EFI_UNSUPPORTED**. If there is no media in the device, the function returns **EFI_NO_MEDIA**. If the **MediaId** is not the ID for the current media in the device, the function returns **EFI_MEDIA_CHANGED**.

If the security protocol fails to complete within the **Timeout** period, the function shall return **EFI_TIMEOUT**.

If the security protocol command completes without an error, the function shall return **EFI_SUCCESS**. If the security protocol command completes with an error, the function shall return **EFI_DEVICE_ERROR**.
## Status Codes Returned

<table>
<thead>
<tr>
<th>Code</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>EFI_SUCCESS</td>
<td>The security protocol command completed successfully.</td>
</tr>
<tr>
<td>EFI_WARN_BUFFER_TOO_SMALL</td>
<td>The PayloadBufferSize was too small to store the available data from the device. The PayloadBuffer contains the truncated data.</td>
</tr>
<tr>
<td>EFI_UNSUPPORTED</td>
<td>The given MediaId does not support security protocol commands.</td>
</tr>
<tr>
<td>EFI_DEVICE_ERROR</td>
<td>The security protocol command completed with an error.</td>
</tr>
<tr>
<td>EFI_NO_MEDIA</td>
<td>There is no media in the device.</td>
</tr>
<tr>
<td>EFI_MEDIA_CHANGED</td>
<td>The MediaId is not for the current media.</td>
</tr>
<tr>
<td>EFI_INVALID_PARAMETER</td>
<td>The PayloadBuffer or PayloadTransferSize is NULL and PayloadBufferSize is non-zero.</td>
</tr>
<tr>
<td>EFI_TIMEOUT</td>
<td>A timeout occurred while waiting for the security protocol command to execute.</td>
</tr>
</tbody>
</table>

### EFI_STORAGE_SECURITY_COMMAND_PROTOCOL.SendData()

#### Summary
Send a security protocol command to a device.

#### Prototype

```c
typedef EFI_STATUS (EFIAPI *EFI_STORAGE_SECURITY_SEND_DATA) (  
  IN EFI_STORAGE_SECURITY_COMMAND_PROTOCOL  
  *This,  
  IN UINT32 MediaId,  
  IN UINT64 Timeout,  
  IN UINT8 SecurityProtocolId,  
  IN UINT16 SecurityProtocolSpecificData,  
  IN UINTN PayloadBufferSize,  
  IN VOID *PayloadBuffer  
);```

#### Parameters

- **This**: Indicates a pointer to the calling context. Type **EFI_STORAGE_SECURITY_COMMAND_PROTOCOL** is defined in the **EFI_STORAGE_SECURITY_COMMAND_PROTOCOL** description.
- **MediaId**: ID of the medium to send data to. If there is no block IO protocol supported by the physical device, the value of MediaId is undefined.
- **Timeout**: The timeout, in 100ns units, to use for the execution of the security protocol command. A **Timeout** value of 0 means that this function will wait indefinitely for the security protocol command to execute. If **Timeout** is greater than zero, then this function will return **EFI_TIMEOUT** if the time required to execute the receive data command is greater than **Timeout**.
**SecurityProtocolId**
The value of the “Security Protocol” parameter of the security protocol command to be sent.

**SecurityProtocolSpecificData**
The value of the “Security Protocol Specific” parameter of the security protocol command to be sent.

**PayloadBufferSize**
Size in bytes of the payload data buffer.

**PayloadBuffer**
A pointer to a buffer containing the security protocol command specific payload data for the security protocol command.

**Description**
The `SendData` function sends a security protocol command containing the payload `PayloadBuffer` to the given `MediaId`. The security protocol command sent is defined by `SecurityProtocolId` and contains the security protocol specific data `SecurityProtocolSpecificData`. If the underlying protocol command requires a specific padding for the command payload, the `SendData` function shall add padding bytes to the command payload to satisfy the padding requirements.

For devices supporting the SCSI command set, the security protocol command is sent using the SECURITY PROTOCOL OUT command defined in SPC-4.

For devices supporting the ATA command set, the security protocol command is sent using one of the TRUSTED SEND commands defined in ATA8-ACS if `PayloadBufferSize` is non-zero. If the `PayloadBufferSize` is zero, the security protocol command is sent using the Trusted Non-Data command defined in ATA8-ACS.

If `PayloadBuffer` is `NULL` and `PayloadBufferSize` is non-zero, the function shall return `EFI_INVALID_PARAMETER`.

If the given `MediaId` does not support security protocol commands, the function shall return `EFI_UNSUPPORTED`. If there is no media in the device, the function returns `EFI_NO_MEDIA`. If the `MediaId` is not the ID for the current media in the device, the function returns `EFI_MEDIA_CHANGED`.

If the security protocol fails to complete within the `Timeout` period, the function shall return `EFI_TIMEOUT`.

If the security protocol command completes without an error, the function shall return `EFI_SUCCESS`. If the security protocol command completes with an error, the function shall return `EFI_DEVICE_ERROR`. 
13.15 NVM Express Pass Through Protocol

**EFI_NVM_EXPRESS_PASS_THRU_PROTOCOL**

This section provides a detailed description of the **EFI_NVM_EXPRESS_PASS_THRU_PROTOCOL**.

**Summary**

This protocol provides services that allow NVM Express commands to be sent to an NVM Express controller or to a specific namespace in a NVM Express controller. This protocol interface is optimized for storage.

**GUID**

```c
#define EFI_NVM_EXPRESS_PASS_THRU_PROTOCOL_GUID  
{ 0x52c78312, 0x8edc, 0x4233, 
{ 0x98, 0xf2, 0x1a, 0x1a, 0xa5, 0xe3, 0x88, 0xa5 } }; 
```

**Protocol Interface Structure**

```c
typedef struct _EFI_NVM_EXPRESS_PASS_THRU_PROTOCOL { 
  EFI_NVM_EXPRESS_PASS_THRU_MODE               *Mode; 
  EFI_NVM_EXPRESS_PASS_THRU_PASSTHRU PassThru; 
  EFI_NVM_EXPRESS_PASS_THRU_GET_NEXT_NAMESPACE GetNextNamespace; 
  EFI_NVM_EXPRESS_PASS_THRU_BUILD_DEVICE_PATH  BuildDevicePath; 
  EFI_NVM_EXPRESS_PASS_THRU_GET_NAMESPACE      GetNamespace; 
} EFI_NVM_EXPRESS_PASS_THRU_PROTOCOL; 
```

**Parameters**

- **Mode**

  A pointer to the **EFI_NVM_EXPRESS_PASS_THRU_MODE** data for this NVM Express controller. **EFI_NVM_EXPRESS_PASS_THRU_MODE** is defined in “Related Definitions” below.

- **PassThru**

  Sends an NVM Express Command Packet to an NVM Express controller. See the **EFI_NVM_EXPRESS_PASS_THRU_PROTOCOL.PassThru()** function description.

---

### Status Codes Returned

<table>
<thead>
<tr>
<th>Code</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>EFI_SUCCESS</td>
<td>The security protocol command completed successfully.</td>
</tr>
<tr>
<td>EFI_UNSUPPORTED</td>
<td>The given <strong>MediaId</strong> does not support security protocol commands.</td>
</tr>
<tr>
<td>EFI_DEVICE_ERROR</td>
<td>The security protocol command completed with an error.</td>
</tr>
<tr>
<td>EFI_NO_MEDIA</td>
<td>There is no media in the device.</td>
</tr>
<tr>
<td>EFI_MEDIA_CHANGED</td>
<td>The <strong>MediaId</strong> is not for the current media.</td>
</tr>
<tr>
<td>EFI_INVALID_PARAMETER</td>
<td>The <strong>PayloadBuffer</strong> is <strong>NULL</strong> and <strong>PayloadBufferSize</strong> is non-zero.</td>
</tr>
<tr>
<td>EFI_TIMEOUT</td>
<td>A timeout occurred while waiting for the security protocol command to execute.</td>
</tr>
</tbody>
</table>
GetNextNamespace  Retrieves the next namespace ID for this NVM Express controller. See the
EFI_NVM_EXPRESS_PASS_THRU_PROTOCOL.GetNextNamespace() function description.

BuildDevicePath  Allocates and builds a device path node for a namespace on an NVM Express controller. See the
EFI_NVM_EXPRESS_PASS_THRU_PROTOCOL.BuildDevicePath() function description.

GetNamespace  Translates a device path node to a namespace ID. See the
EFI_NVM_EXPRESS_PASS_THRU_PROTOCOL.GetNamespace() function description.

The following data values in the EFI_NVM_EXPRESS_PASS_THRU_MODE interface are read-only.

Attributes  Additional information on the attributes of the NVM Express controller. See “Related Definitions” below for the list of possible attributes.

IoAlign  Supplies the alignment requirement for any buffer used in a data transfer. IoAlign values of 0 and 1 mean that the buffer can be placed anywhere in memory. Otherwise, IoAlign must be a power of 2, and the requirement is that the start address of a buffer must be evenly divisible by IoAlign with no remainder.

NvmeVersion  Indicates the version of the NVM Express specification that the controller implementation supports. The format of this field is defined in the Version field of the Controller Registers in the NVM Express Specification.

Related Definitions

typedef struct {
  UINT32 Attributes;
  UINT32 IoAlign;
  UINT32 NvmeVersion;
} EFI_NVM_EXPRESS_PASS_THRU_MODE;

#define EFI_NVM_EXPRESS_PASS_THRU_ATTRIBUTES_PHYSICAL     0x0001
#define EFI_NVM_EXPRESS_PASS_THRU_ATTRIBUTES_LOGICAL      0x0002
#define EFI_NVM_EXPRESS_PASS_THRU_ATTRIBUTES_NONBLOCKIO   0x0004
#define EFI_NVM_EXPRESS_PASS_THRU_ATTRIBUTES_CMD_SET_NVM  0x0008

EFI_NVM_EXPRESS_PASS_THRU_ATTRIBUTES_PHYSICAL
  If this bit is set, then the EFI_NVM_EXPRESS_PASS_THRU_PROTOCOL interface is for directly addressable namespaces.

EFI_NVM_EXPRESS_PASS_THRU_ATTRIBUTES_LOGICAL
  If this bit is set, then the EFI_NVM_EXPRESS_PASS_THRU_PROTOCOL interface is for a single volume logical namespace comprised of multiple namespaces.
Description

The **EFI_NVM_EXPRESS_PASS_THRU_PROTOCOL** provides information about an NVM Express controller and the ability to send NVM Express commands to an NVM Express controller or to a specific namespace in a NVM Express controller.

The printable name for the NVM Express controller can be provided through the **EFI_COMPONENT_NAME_PROTOCOL** and the **EFI_COMPONENT_NAME2_PROTOCOL** for multiple languages.

The **Attributes** field of the **EFI_NVM_EXPRESS_PASS_THRU_PROTOCOL** interface tells if the interface is for physical NVM Express controllers or logical NVM Express controllers. Drivers for non-RAID NVM Express controllers will set both the **EFI_NVM_EXPRESS_PASS_THRU_ATTRIBUTES_PHYSICAL**, and the **EFI_NVM_EXPRESS_PASS_THRU_ATTRIBUTES_LOGICAL** bits.

Drivers for RAID controllers that allow access to the physical controllers and logical controllers will produce two **EFI_NVM_EXPRESS_PASS_THRU_PROTOCOL** interfaces: one with the just the **EFI_NVM_EXPRESS_PASS_THRU_ATTRIBUTES_PHYSICAL** bit set and another with just the **EFI_NVM_EXPRESS_PASS_THRU_ATTRIBUTES_LOGICAL** bit set. One interface can be used to access the physical controllers attached to the RAID controller, and the other can be used to access the logical controllers attached to the RAID controller for its current configuration.

Drivers for RAID controllers that do not allow access to the physical controllers will produce one **EFI_NVM_EXPRESS_PASS_THROUGH_PROTOCOL** interface with just the **EFI_NVM_EXPRESS_PASS_THRU_ATTRIBUTES_LOGICAL** bit set. The interface for logical controllers can also be used by a file system driver to mount the RAID volumes. An **EFI_NVM_EXPRESS_PASS_THRU_PROTOCOL** with neither **EFI_NVM_EXPRESS_PASS_THRU_ATTRIBUTES_LOGICAL** nor **EFI_NVM_EXPRESS_PASS_THRU_ATTRIBUTES_PHYSICAL** set is an illegal configuration.

The **Attributes** field also contains the **EFI_NVM_EXPRESS_PASS_THRU_ATTRIBUTES_NONBLOCKIO** bit. All **EFI_NVM_EXPRESS_PASS_THRU_PROTOCOL** interfaces must support blocking I/O. If this bit is set, then the interface supports both blocking I/O and non-blocking I/O.

The **Attributes** field also contains the **EFI_NVM_EXPRESS_PASS_THRU_ATTRIBUTES_CMD_SET_NVM** bit. If this bit is set, the controller supports the NVM Express command set.

Each **EFI_NVM_EXPRESS_PASS_THRU_PROTOCOL** instance must have an associated device path. Typically this will have an ACPI device path node and a PCI device path node, although variation will exist.
**EFI_NVM_EXPRESS_PASS_THRU_PROTOCOL.PassThru()**

**Summary**
Sends an NVM Express Command Packet to an NVM Express controller or namespace. This function supports both blocking I/O and non-blocking I/O. The blocking I/O functionality is required, and the non-blocking I/O functionality is optional.

**Prototype**
```c
typedef EFI_STATUS
   (EFIAPI *EFI_NVM_EXPRESS_PASS_THRU_PASSTHRU) (
   IN EFI_NVM_EXPRESS_PASS_THRU_PROTOCOL *This,
   IN UINT32 NamespaceId,
   IN OUT EFI_NVM_EXPRESS_PASS_THRU_COMMAND_PACKET *Packet,
   IN EFI_EVENT Event OPTIONAL);
```

**Parameters**
- **This** A pointer to theEFI_NVM_EXPRESS_PASS_THRU_PROTOCOL instance. TypeEFI_NVM_EXPRESS_PASS_THRU_PROTOCOLis defined in Section 13.15, above.
- **NamespaceId** A 32 bit namespace ID as defined in the NVMe specification to which the NVM Express Command Packet will be sent. A value of 0 denotes the NVM Express controller, a value of all 0xFF’s (all bytes are 0xFF) in the namespace ID specifies that the command packet should be sent to all valid namespaces.
- **Packet** A pointer to the NVM Express Command Packet. See “Related Definitions” below for a description ofEFI_NVM_EXPRESS_PASS_THRU_COMMAND_PACKET.
- **Event** If non-blocking I/O is not supported then Event is ignored, and blocking I/O is performed. If Event is NULL, then blocking I/O is performed. If Event is not NULL and non-blocking I/O is supported, then non-blocking I/O is performed, and Event will be signaled when the NVM Express Command Packet completes.
typedef struct {
    UINT64 CommandTimeout;
    VOID *TransferBuffer OPTIONAL;
    UINT32 TransferLength OPTIONAL;
    VOID *MetaDataBuffer OPTIONAL;
    UINT32 MetadataLength OPTIONAL;
    UINT8 QueueType;
    EFI_NVM_EXPRESS_COMMAND *NvmeCmd;
    EFI_NVM_EXPRESS_COMPLETION *NvmeCompletion;
} EFI_NVM_EXPRESS_PASS_THRU_COMMAND_PACKET;

CommandTimeout: The timeout in 100 ns units to use for the execution of this NVM Express Command Packet. A Timeout value of 0 means that this function will wait indefinitely for the command to execute. If Timeout is greater than zero, then this function will return EFI_TIMEOUT if the time required to execute the NVM Express command is greater than Timeout.

TransferBuffer: A pointer to the data buffer to transfer between the host and the NVM Express controller for read, write, and bi-directional commands. For all write and non-data commands where TransferLength is 0 this field is optional and may be NULL. If this field is not NULL, then it must be aligned on the boundary specified by the IoAlign field in the EFI_NVM_EXPRESS_PASS_THRU_MODE structure.

TransferLength: On input, the size in bytes of TransferBuffer. On output, the number of bytes transferred to or from the NVM Express controller or namespace.

MetadataBuffer: A pointer to the optional metadata buffer to transfer between the host and the NVM Express controller. For all commands where no metadata is transferred between the host and the controller, this field is optional and may be NULL. If this field is not NULL, then it must be aligned on the boundary specified by the IoAlign field in the EFI_NVM_EXPRESS_PASS_THRU_MODE structure.

MetadataLength: On input, the size in bytes of MetadataBuffer. On output, the number of bytes transferred to or from the NVM Express controller or namespace.

QueueType: The type of the queue that the NVMe command should be posted to. A value of 0 indicates it should be posted to the Admin Submission Queue. A value of 1 indicates it should be posted to an I/O Submission Queue.

NvmeCmd: A pointer to an NVM Express Command Packet.

NvmeCompletion: The raw NVM Express completion queue entry as defined in the NVM Express Specification.
Description

The **PassThru()** function sends the NVM Express Command Packet specified by **Packet** to the NVM Express controller. If the driver supports non-blocking I/O and **Event** is not NULL, then the driver will return immediately after the command is sent to the selected controller, and will later signal **Event** when the command has completed.

If the driver supports non-blocking I/O and **Event** is NULL, then the driver will send the command to the selected device and block until it is complete.

If the driver does not support non-blocking I/O, then the **Event** parameter is ignored, and the driver will send the command to the selected device and block until it is complete.

If **Packet** is successfully sent to the NVM Express controller, then **EFI_SUCCESS** is returned.

If a device error occurs while sending the **Packet**, then **EFI_DEVICE_ERROR** is returned.

If a timeout occurs during the execution of **Packet**, then **EFI_TIMEOUT** is returned.

If **NamespaceId** is invalid for the NVM Express controller, then **EFI_INVALID_PARAMETER** is returned.

If **TransferBuffer** or **MetadataBuffer** do not meet the alignment requirement specified by the **IoAlign** field of the **EFI_NVM_EXPRESS_PASS_THRU_MODE** structure, then **EFI_INVALID_PARAMETER** is returned. If the **QueueType** is not 0 (Admin Submission Queue) or 1 (I/O Submission Queue), then **EFI_INVALID_PARAMETER** is returned. If any of the other fields of **Packet** are invalid, then **EFI_INVALID_PARAMETER** is returned.

If the data buffer described by **TransferBuffer** and **TransferLength** is too big to be transferred in a single command, then no data is transferred and **EFI_BAD_BUFFER_SIZE** is returned. The number of bytes that can be transferred in a single command are returned in **TransferLength**.

If **EFI_SUCCESS**, **EFI_DEVICE_ERROR**, or **EFI_TIMEOUT** is returned, then the caller must examine the **NvmeCompletion** field in **Packet**.

If non-blocking I/O is being used, then the **NvmeCompletion** field in **Packet** will not be valid until the **Event** associated with **Packet** is signaled.

If **EFI_NOT_READY**, **EFI_INVALID_PARAMETER**, **EFI_BAD_BUFFER_SIZE**, or **EFI_UNSUPPORTED** is returned, then **Packet** was never sent, so the **NvmeCompletion** field in **Packet** is not valid. If non-blocking I/O is being used, the **Event** associated with **Packet** will not be signaled.
Status Codes Returned

<table>
<thead>
<tr>
<th>Code</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>EFI_SUCCESS</td>
<td>The NVM Express Command Packet was sent by the host. TransferLength bytes were transferred to or from TransferBuffer. See NvmeCompletion (above) for additional status information.</td>
</tr>
<tr>
<td>EFI_BAD_BUFFER_SIZE</td>
<td>The NVM Express Command Packet was not executed. The number of bytes that could be transferred is returned in TransferLength.</td>
</tr>
<tr>
<td>EFI_NOT_READY</td>
<td>The NVM Express Command Packet could not be sent because the controller is not ready. The caller may retry again later.</td>
</tr>
<tr>
<td>EFI_DEVICE_ERROR</td>
<td>A device error occurred while attempting to send the NVM Express Command Packet. See NvmeCompletion (above) for additional status information.</td>
</tr>
<tr>
<td>EFI_INVALID_PARAMETER</td>
<td>NamespaceId or the contents of EFI_NVM_EXPRESS_PASS_THRU_COMMAND_PACKET are invalid. The NVM Express Command Packet was not sent, so no additional status information is available.</td>
</tr>
<tr>
<td>EFI_UNSUPPORTED</td>
<td>The command described by the NVM Express Command Packet is not supported by the NVM Express controller. The NVM Express Command Packet was not sent so no additional status information is available.</td>
</tr>
<tr>
<td>EFI_TIMEOUT</td>
<td>A timeout occurred while waiting for the NVM Express Command Packet to execute. See NvmeCompletion (above) for additional status information.</td>
</tr>
</tbody>
</table>

Related Definitions

typedef struct {
    UINT32 OpCode : 8;
    UINT32 FusedOperation : 2;
    UINT32 Reserved : 22;
} NVME_CDW0;

//***************************************************************
// FusedOperation
//***************************************************************
#define NORMAL_CMD 0x00
#define FUSED_FIRST_CMD 0x01
#define FUSED_SECOND_CMD 0x02

typedef struct {
    NVME_CDW0 Cdw0;
    UINT8 Flags;
    UINT32 Nsid;
    UINT32 Cdw2;
    UINT32 Cdw3;
    UINT32 Cdw10;
    UINT32 Cdw11;
    UINT32 Cdw12;
}
**UEFI Specification, Version 2.9**

**Protocols — Media Access**

```c
UINT32 Cdw13;
UINT32 Cdw14;
UINT32 Cdw15;
} EFI_NVM_EXPRESS_COMMAND;

// **********************************************
// Flags
// **********************************************
#define CDW2_VALID0x01
#define CDW3_VALID0x02
#define CDW10_VALID0x04
#define CDW11_VALID0x08
#define CDW12_VALID0x10
#define CDW13_VALID0x20
#define CDW14_VALID0x40
#define CDW15_VALID0x80

// This structure maps to the NVM Express specification Completion Queue Entry
typedef struct {
    UINT32 DW0;
    UINT32 DW1;
    UINT32 DW2;
    UINT32 DW3;
} EFI_NVM_EXPRESS_COMPLETION;

**EFI_NVM_EXPRESS_PASS_THRU_PROTOCOL.GetNextNamespace()**

**Summary**

Used to retrieve the next namespace ID for this NVM Express controller.

**Prototype**

```c
typedef
EFI_STATUS
(EIFIAPI *EFI_NVM_EXPRESS_PASS_THRU_GET_NEXT_NAMESPACE) ( 
    IN EFI_NVM_EXPRESS_PASS_THRU_PROTOCOL *This,
    IN OUT UINT32 *NamespaceId
);
```

**Parameters**

- **This** A pointer to the **EFI_NVM_EXPRESS_PASS_THRU_PROTOCOL** instance.
NamespaceId

On input, a pointer to a valid namespace ID on this NVM Express controller or a pointer to the value 0xFFFFFFFF. A pointer to the value 0xFFFFFFFF retrieves the first valid namespace ID on this NVM Express controller.

Description

The EFI_NVM_EXPRESS_PASS_THRU_PROTOCOL.GetNextNamespace() function retrieves the next valid namespace ID on this NVM Express controller. If on input the value pointed to by NamespaceId is 0xFFFFFFFF, then the first valid namespace ID defined on the NVM Express controller is returned in the location pointed to by NamespaceId and a status of EFI_SUCCESS is returned.

If on input the value pointed to by NamespaceId is an invalid namespace ID other than 0xFFFFFFFF, then EFI_INVALID_PARAMETER is returned.

If on input the value pointed to by NamespaceId is a valid namespace ID, then the next valid namespace ID on the NVM Express controller is returned in the location pointed to by NamespaceId, and EFI_SUCCESS is returned.

If the value pointed to by NamespaceId is the namespace ID of the last namespace on the NVM Express controller, then EFI_NOT_FOUND is returned.

Status Codes Returned

<table>
<thead>
<tr>
<th>Status Code</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>EFI_SUCCESS</td>
<td>The Namespace ID of the next Namespace was returned.</td>
</tr>
<tr>
<td>EFI_NOT_FOUND</td>
<td>There are no more namespaces defined on this controller.</td>
</tr>
<tr>
<td>EFI_INVALID_PARAMETER</td>
<td>NamespaceId is an invalid value other than 0xFFFFFFFF.</td>
</tr>
</tbody>
</table>

EFI_NVM_EXPRESS_PASS_THRU_PROTOCOL.BuildDevicePath()

Summary

Used to allocate and build a device path node for an NVM Express namespace on an NVM Express controller.

Prototype

```c
typedef EFI_STATUS (EFIoxetine *EFI_NVM_EXPRESS_PASS_THRU_BUILD_DEVICE_PATH) (IN EFI_NVM_EXPRESS_PASS_THRU_PROTOCOL *This,
                          IN UINT32 NamespaceId,
                          OUT EFI_DEVICE_PATH_PROTOCOL **DevicePath);
```

Parameters

- **This**
  - A pointer to the EFI_NVM_EXPRESS_PASS_THRU_PROTOCOL instance. Type EFI_NVM_EXPRESS_PASS_THRU_PROTOCOL is defined in Section 13.15.

- **NamespaceId**
  - The NVM Express namespace ID for which a device path node is to be allocated and built.
DevicePath

A pointer to a single device path node that describes the NVM Express namespace specified by NamespaceId. This function is responsible for allocating the buffer DevicePath with the boot service AllocatePool(). It is the caller’s responsibility to free DevicePath when the caller is finished with DevicePath.

Description

The EFI_NVM_EXPRESS_PASS_THRU_PROTOCOL.BuildDevicePath() function allocates and builds a single device path node for the NVM Express namespace specified by NamespaceId. If the NamespaceId is not valid, then EFI_NOT_FOUND is returned. If DevicePath is NULL, then EFI_INVALID_PARAMETER is returned. If there are not enough resources to allocate the device path node, then EFI_OUT_OF_RESOURCES is returned. Otherwise, DevicePath is allocated with the boot service AllocatePool(), the contents of DevicePath are initialized to describe the NVM Express namespace specified by NamespaceId, and EFI_SUCCESS is returned.

Status Codes Returned

<table>
<thead>
<tr>
<th>Code</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>EFI_SUCCESS</td>
<td>The device path node that describes the NVM Express namespace specified by</td>
</tr>
<tr>
<td></td>
<td>NamespaceId was allocated and returned in DevicePath.</td>
</tr>
<tr>
<td>EFI_NOT_FOUND</td>
<td>The NamespaceId is not valid.</td>
</tr>
<tr>
<td>EFI_INVALID_PARAMETER</td>
<td>DevicePath is NULL.</td>
</tr>
<tr>
<td>EFI_OUT_OF_RESOURCES</td>
<td>There are not enough resources to allocate the DevicePath node.</td>
</tr>
</tbody>
</table>

EFI_NVM_EXPRESS_PASS_THRU_PROTOCOL.GetNamespace()

Summary

Used to translate a device path node to a namespace ID.

Prototype

```c
typedef EFI_STATUS (EFIAPI *EFI_NVM_EXPRESS_PASS_THRU_GET_NAMESPACE) (  
  IN EFI_NVM_EXPRESS_PASS_THRU_PROTOCOL *This,  
  IN EFI_DEVICE_PATH_PROTOCOL *DevicePath,  
  OUT UINT32 *NamespaceId  
);
```

Parameters

- **This**  
  A pointer to the EFI_NVM_EXPRESS_PASS_THRU_PROTOCOL instance. Type EFI_NVM_EXPRESS_PASS_THRU_PROTOCOL is defined in Section 13.15.

- **DevicePath**  
  A pointer to the device path node that describes an NVM Express namespace on the NVM Express controller.

- **NamespaceId**  
  The NVM Express namespace ID contained in the device path node.
Description
The `EFI_NVM_EXPRESS_PASS_THRU_PROTOCOL.GetNamespace()` function determines the namespace ID associated with the namespace described by `DevicePath`. If `DevicePath` is a device path node type that the NVM Express Pass Thru driver supports, then the NVM Express Pass Thru driver will attempt to translate the contents `DevicePath` into a namespace ID. If this translation is successful, then that namespace ID is returned in `NamespaceId`, and `EFI_SUCCESS` is returned.

Status Codes Returned

<table>
<thead>
<tr>
<th>Status Code</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>EFI_INVALID_PARAMETER</td>
<td>If <code>DevicePath</code> or <code>NamespaceId</code> are NULL, then <code>EFI_INVALID_PARAMETER</code> is returned.</td>
</tr>
<tr>
<td>EFI_UNSUPPORTED</td>
<td>If <code>DevicePath</code> is not a device path node type that the NVM Express Pass Thru driver supports, then <code>EFI_UNSUPPORTED</code> is returned.</td>
</tr>
<tr>
<td>EFI_NOT_FOUND</td>
<td>If <code>DevicePath</code> is a device path node type that the NVM Express Pass Thru driver supports, but there is not a valid translation from <code>DevicePath</code> to a namespace ID, then <code>EFI_NOT_FOUND</code> is returned.</td>
</tr>
</tbody>
</table>

13.16 SD MMC Pass Thru Protocol

**EFI_SD_MMC_PASS_THRU_PROTOCOL**

This section provides a detailed description of the `EFI_SD_MMC_PASS_THRU_PROTOCOL`.

The protocol provides services that allow SD/eMMC commands to be sent to an SD/eMMC controller. All interfaces and definitions from this section apply equally to SD and eMMC controllers.

For the sake of brevity, the rest of this section refers only to SD devices and controllers and does not specifically mention eMMC devices and controllers.

**GUID**

```c
#define EFI_SD_MMC_PASS_THRU_PROTOCOL_GUID \
{ 0x716ef0d9, 0xff83, 0x4f69, \
{ 0x81, 0xe9, 0x51, 0x8b, 0xd3, 0x9a, 0x8e, 0x70 } }
```

**Protocol Interface Structure**

```c
typedef struct _EFI_SD_MMC_PASS_THRU_PROTOCOL {
    UINTN                                    IoAlign
    EFI_SD_MMC_PASS_THRU_PASSTHRU            PassThru;
    EFI_SD_MMC_PASS_THRU_GET_NEXT_SLOT       GetNextSlot;
    EFI_SD_MMC_PASS_THRU_BUILD_DEVICE_PATH   BuildDevicePath;
    EFI_SD_MMC_PASS_THRU_GET_SLOT_NUMBER     GetSlotNumber;
    EFI_SD_MMC_PASS_THRU_RESET_DEVICE        ResetDevice;
} EFI_SD_MMC_PASS_THRU_PROTOCOL;
```

**Parameters**

- **IoAlign** Supplies the alignment requirement for any buffer used in a data transfer. `IoAlign` values of 0 and 1 mean that the buffer can be placed anywhere in memory. Otherwise, `IoAlign` must be a power
of 2, and the requirement is that the start address of a buffer must be evenly divisible by \texttt{IoAlign} with no remainder.

**PassThru** Sends SD command to the SD controller. See the \texttt{PassThru()} function description.

**GetNextSlot** Retrieves a next slot on an SD controller. See the \texttt{GetNextSlot()} function description.

**BuildDevicePath** Allocates and builds a device path node for an SD card on the SD controller. See the \texttt{BuildDevicePath()} function description.

**GetSlotNumber** Retrieves the SD card slot number based on the input device path. See the \texttt{GetSlotNumber()} function description.

**ResetDevice** Resets an SD card connected to the SD controller. See the \texttt{ResetDevice()} function description.

---

**\texttt{EFI_SD_MMC_PASS_THRU_PROTOCOL.PassThru()}**

**Summary**
Sends SD command to an SD card that is attached to the SD controller.

**Prototype**

```c
typedef EFI_STATUS (EFIAPI *\texttt{EFI_SD_MMC_PASS_THRU_PASSTHRU}) (
    IN EFI_SD_MMC_PASS_THRU_PROTOCOL *This,
    IN UINT8 Slot,
    IN OUT EFI_SD_MMC_PASS_THRU_COMMAND_PACKET *Packet,
    IN EFI_EVENT Event OPTIONAL
);
```

**Parameters**

- \texttt{This} A pointer to the \texttt{EFI_SD_MMC_PASS_THRU_PROTOCOL} instance.
- \texttt{Slot} The slot number of the SD card to send the command to.
- \texttt{Packet} A pointer to the SD command data structure. See “Related Definitions” below for a description of \texttt{EFI_SD_MMC_PASS_THRU_COMMAND_PACKET}.
- \texttt{Event} If non-blockingI/O is not supported then \texttt{Event} is ignored, and blocking I/O is performed. If \texttt{Event} is \texttt{NULL}, then blockingI/O is performed. If \texttt{Event} is not \texttt{NULL} and non-blockingI/O is supported, then non-blockingI/O is performed, and \texttt{Event} will be signaled when the SDcommand completes.
typedef struct {
    EFI_SD_MMC_COMMAND_BLOCK       *SdMmcCmdBlk;
    EFI_SD_MMC_STATUS_BLOCK        *SdMmcStatusBlk;
    UINT64                          Timeout;
    VOID                           *InDataBuffer;
    VOID                           *OutDataBuffer;
    UINT32                          InTransferLength;
    UINT32                          OutTransferLength;
    EFI_STATUS                      TransactionStatus;
} EFI_SD_MMC_PASS_THRU_COMMAND_PACKET;

SdMmcCmdBlk A pointer to a command specific data buffer allocated and filled by the caller. See “Related Definitions” below for a description of EFI_SD_MMC_COMMAND_BLOCK.

SdMmcStatusBlk A pointer to a command specific response data buffer allocated by the caller and filled by the PassThru function. See “Related Definitions” below for a description of EFI_SD_MMC_STATUS_BLOCK.

Timeout The timeout, in 100 ns units, to use for the execution of this SDcommand. A Timeout value of 0 means that this function will wait indefinitely for the SDcommand to execute. If Timeout is greater than zero, then this function will return EFI_TIMEOUT if the time required to execute the SDcommand is greater than Timeout.

InDataBuffer A pointer to a buffer for the data transferred from the SD card during processing of read and bidirectional commands. For all write and non-data commands this field is optional and may be NULL.

OutDataBuffer A pointer to a buffer for the data to be transferred to the SD card during processing of write or bidirectional commands. For all read and non-data commands this field is optional and may be NULL.

InTransferLength On input, the size, in bytes, of InDataBuffer. On output, the number of bytes transferred between the SD controller and the SD device. If InTransferLength is larger than the SD controller can handle, no data will be transferred, InTransferLength will be updated to contain the number of bytes that the SD controller is able to transfer, and EFI_BAD_BUFFER_SIZE will be returned.

OutTransferLength On input, the size, in bytes of OutDataBuffer. On output, the number of bytes transferred between SD Controller and the SD device. If OutTransferLength is larger than the SD controller can handle, no data will be transferred. OutTransferLength will be updated to contain the number of bytes that the SD controller is able to transfer, and EFI_BAD_BUFFER_SIZE will be returned.

TransactionStatus Transaction status. When PathThru() function is used in a blocking mode, the status must be the same as the status returned by the PathThru() function. When PathThru() function is used in a non-blocking mode, the field is updated with the transaction status once transaction is completed.
Related Definitions

typedef struct {
    UINT16   CommandIndex;
    UINT32   CommandArgument;
    UINT32   CommandType; // One of the EFI_SD_MMC_COMMAND_TYPE values
    UINT32   ResponseType; // One of the EFI_SD_MMC_RESPONSE_TYPE values
} EFI_SD_MMC_COMMAND_BLOCK;

typedef struct {
    UINT32   Resp0;
    UINT32   Resp1;
    UINT32   Resp2;
    UINT32   Resp3;
} EFI_SD_MMC_STATUS_BLOCK;

typedef enum {
    SdMmcCommandTypeBc, // Broadcast commands, no response
    SdMmcCommandTypeBcr, // Broadcast commands with response
    SdMmcCommandTypeAc,  // Addressed(point-to-point) commands
    SdMmcCommandTypeAdtc // Addressed(point-to-point) data transfer
                         // commands
} EFI_SD_MMC_COMMAND_TYPE;

typedef enum {
    SdMmcResponseTypeR1,
    SdMmcResponseTypeR1b,
    SdMmcResponseTypeR2,
    SdMmcResponseTypeR3,
    SdMmcResponseTypeR4,
    SdMmcResponseTypeR5,
    SdMmcResponseTypeR5b,
    SdMmcResponseTypeR6,
    SdMmcResponseTypeR7
} EFI_SD_MMC_RESPONSE_TYPE;

Description

The PassThru() function sends the SD command specified by Packet to the SD card specified by Slot.

If Packet is successfully sent to the SD card, then EFI_SUCCESS is returned. If a device error occurs while sending the Packet, then EFI_DEVICE_ERROR is returned. If Slot is not in a valid range for the SD controller, then EFI_INVALID_PARAMETER is returned. If Packet defines a data command but both InDataBuffer and OutDataBuffer are NULL, EFI_INVALID_PARAMETER is returned.
Status Codes Returned

<table>
<thead>
<tr>
<th>Code</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>EFI_SUCCESS</td>
<td>The SD Command Packet was sent by the host.</td>
</tr>
<tr>
<td>EFI_DEVICE_ERROR</td>
<td>A device error occurred while attempting to send the SD command Packet.</td>
</tr>
<tr>
<td>EFI_INVALID_PARAMETER</td>
<td>Packet, Slot, or the contents of the Packet is invalid.</td>
</tr>
<tr>
<td>EFI_INVALID_PARAMETER</td>
<td>Packet defines a data command but both InDataBuffer and OutDataBuffer are NULL.</td>
</tr>
<tr>
<td>EFI_NO_MEDIA</td>
<td>SD Device not present in the Slot.</td>
</tr>
<tr>
<td>EFI_UNSUPPORTED</td>
<td>The command described by the SD Command Packet is not supported by the host controller.</td>
</tr>
<tr>
<td>EFI_BAD_BUFFER_SIZE</td>
<td>The InTransferLength or OutTransferLength exceeds the limit supported by SD card (i.e. if the number of bytes exceed the Last LBA).</td>
</tr>
<tr>
<td>EFI_DEVICE_ERROR</td>
<td>The command was not sent due to a device error</td>
</tr>
</tbody>
</table>

**EFI_SD_MMC_PASS_THRU_PROTOCOL.GetNextSlot()**

**Summary**

Used to retrieve next slot numbers supported by the SD controller. The function returns information about all available slots (populated or not-populated).

**Prototype**

```c
typedef EFI_STATUS
    (EFIAPI *EFI_SD_MMC_PASS_THRU_GET_NEXT_SLOT) (
        IN EFI_SD_MMC_PASS_THRU_PROTOCOL *This,
        IN OUT UINT8 *Slot
    );
```

**Parameters**

- **This**
  A pointer to the EFI_SD_MMC_PASS_THRU_PROTOCOL instance.
- **Slot**
  On input, a pointer to a slot number on the SD controller. On output, a pointer to the next slot number on the SD controller. An input value of 0xFF retrieves the first slot number on the SD controller.

**Description**

The **GetNextSlot()** function retrieves the next slot number on an SD controller. If on input Slot is 0xFF, then the slot number of the first slot on the SD controller is returned.

If Slot is a slot number that was returned on a previous call to **GetNextSlot()**, then the slot number of the next slot on the SD controller is returned.

If Slot is not 0xFF and Slot was not returned on a previous call to **GetNextSlot()**, **EFI_INVALID_PARAMETER** is returned.

If Slot is the slot number of the last slot on the SD controller, then **EFI_NOT_FOUND** is returned.
Status Codes Returned

<table>
<thead>
<tr>
<th>Code</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>EFI_SUCCESS</td>
<td>The next slot number on the SD controller was returned in Slot.</td>
</tr>
<tr>
<td>EFI_NOT_FOUND</td>
<td>There are no more slots on this SD controller.</td>
</tr>
<tr>
<td>EFI_INVALID_PARAMETER</td>
<td>Slot is not 0xFF and Slot was not returned on a previous call to GetNextSlot().</td>
</tr>
</tbody>
</table>

**EFI_SD_MMC_PASS_THRU_PROTOCOL.BuildDevicePath()**

**Summary**

Used to allocate and build a device path node for an SD card on the SD controller.

**Prototype**

```c
typedef EFI_STATUS (EFIAPI *EFI_SD_MMC_PASS_THRU_BUILD_DEVICE_PATH) (  
    IN EFI_SD_MMC_PASS_THRU_PROTOCOL *This,  
    IN UINT8 Slot,  
    OUT EFI_DEVICE_PATH_PROTOCOL **DevicePath
);
```

**Parameters**

- **This**
  A pointer to the `EFI_SD_MMC_PASS_THRU_PROTOCOL` instance.

- **Slot**
  Specifies the slot number of the SD card for which a device path node is to be allocated and built.

- **DevicePath**
  A pointer to a single device path node that describes the SD card specified by Slot. This function is responsible for allocating the buffer `DevicePath` with the boot service `AllocatePool()`. It is the caller’s responsibility to free `DevicePath` when the caller is finished with `DevicePath`.

**Description**

The `BuildDevicePath()` function allocates and builds a single device node for the SD card specified by `Slot`. If the SD card specified by `Slot` is not present on the SD controller, then `EFI_NOT_FOUND` is returned. If `DevicePath` is `NULL`, then `EFI_INVALID_PARAMETER` is returned. If there are not enough resources to allocate the device path node, then `EFI_OUT_OF_RESOURCES` is returned.

Otherwise, `DevicePath` is allocated with the boot service `AllocatePool()`, the contents of `DevicePath` are initialized to describe the SD card specified by `Slot`, and `EFI_SUCCESS` is returned.
Status Codes Returned

<table>
<thead>
<tr>
<th>Status Code</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>EFI_SUCCESS</td>
<td>The device path node that describes the SD card specified by Slot was allocated and returned in DevicePath.</td>
</tr>
<tr>
<td>EFI_NOT_FOUND</td>
<td>The SD card specified by Slot does not exist on the SD controller</td>
</tr>
<tr>
<td>EFI_INVALID_PARAMETER</td>
<td>DevicePath is NULL</td>
</tr>
<tr>
<td>EFI_OUT_OF_RESOURCES</td>
<td>There are not enough resources to allocate DevicePath</td>
</tr>
</tbody>
</table>

**EFI_SD_MMC_PASS_THRU_PROTOCOL.GetSlotNumber()**

**Summary**
This function retrieves an SD card slot number based on the input device path.

**Prototype**
```c
typedef EFI_STATUS (EFIAPI *EFI_SD_MMC_PASS_THRU_GET_SLOT_NUMBER) (
    IN EFI_SD_MMC_PASS_THRU_PROTOCOL *This,
    IN EFI_DEVICE_PATH_PROTOCOL *DevicePath,
    OUT UINT8 *Slot
);
```

**Parameters**
- **This**
  A pointer to the EFI_SD_MMC_PASS_THRU_PROTOCOL instance.
- **DevicePath**
  A pointer to the device path node that describes a SD card on the SD controller.
- **Slot**
  On return, points to the slot number of an SD card on the SD controller.

**Description**
The GetSlotNumber() function retrieves slot number for the SD card specified by the DevicePath node. If DevicePath is NULL, EFI_INVALID_PARAMETER is returned. If DevicePath is not a device path node type that the SD Pass Thru driver supports, EFI_UNSUPPORTED is returned.

**Status Codes Returned**

<table>
<thead>
<tr>
<th>Status Code</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>EFI_SUCCESS</td>
<td>SD card slot number is returned in Slot.</td>
</tr>
<tr>
<td>EFI_INVALID_PARAMETER</td>
<td>Slot or DevicePath is NULL</td>
</tr>
<tr>
<td>EFI_UNSUPPORTED</td>
<td>DevicePath is not a device path node type that the SD Pass Thru driver supports.</td>
</tr>
</tbody>
</table>

**EFI_SD_MMC_PASS_THRU_PROTOCOL.ResetDevice()**

**Summary**
Resets an SD card that is connected to the SD controller.
Prototype

typedef
EFI_STATUS
(EFI_API * EFI_SD_MMC_PASS_THRU_RESET_DEVICE) (
    IN EFI_SD_MMC_PASS_THRU_PROTOCOL  *This,
    IN UINT8 Slot
);

Parameters

This                    A pointer to the EFI_SD_MMC_PASS_THRU_PROTOCOL instance.
Slot                    Specifies the slot number of the SD card to be reset.

Description

The ResetDevice() function resets the SD card specified by Slot. If this SD controller does not support a device reset operation, EFI_UNSUPPORTED is returned. If Slot is not in a valid slot number for this SD controller, EFI_INVALID_PARAMETER is returned.

If the device reset operation is completed, EFI_SUCCESS is returned.

Status Codes Returned

<table>
<thead>
<tr>
<th>Status Code</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>EFI_SUCCESS</td>
<td>The SD card specified by Slot was reset.</td>
</tr>
<tr>
<td>EFI_UNSUPPORTED</td>
<td>The SD controller does not support a device reset operation.</td>
</tr>
<tr>
<td>EFI_INVALID_PARAMETER</td>
<td>Slot number is invalid.</td>
</tr>
<tr>
<td>EFI_NO_MEDIA</td>
<td>SD Device not present in the Slot.</td>
</tr>
<tr>
<td>EFI_DEVICE_ERROR</td>
<td>The reset command failed due to a device error</td>
</tr>
</tbody>
</table>
13.17 RAM Disk Protocol

EFI_RAM_DISK_PROTOCOL

Summary
RAM disk aware application invokes this protocol to register/unregister a specified RAM disk.

GUID
#define EFI_RAM_DISK_PROTOCOL_GUID
   { 0xab38a0df, 0x6873, 0x44a9, 
     { 0x87, 0xe6, 0xd4, 0xeb, 0x56, 0x14, 0x84, 0x49 }}

Protocol Interface Structure
typedef struct _EFI_RAM_DISK_PROTOCOL {
   EFI_RAM_DISK_REGISTER_RAMDISK Register;
   EFI_RAM_DISK_UNREGISTER_RAMDISK Unregister;
} EFI_RAM_DISK_PROTOCOL;

Members
   Register          Register a RAM disk with specified buffer address, size and type.
   Unregister        Unregister the RAM disk specified by a device path.

Description
This protocol defines a standard interface for UEFI applications, drivers and OS loaders to register/unregister a RAM disk.

The key points are:
   • The consumer of this protocol is responsible for allocating/freeing memory used by RAM Disk if needed and deciding the initial content, as in most scenarios only the consumer knows which type and how much memory should be used.

EFI_RAM_DISK_PROTOCOL.Register()

Summary
This function is used to register a RAM disk with specified address, size and type.
Prototype

typedef
EFI_STATUS
(EIFIAPI *EFI_RAM_DISK_REGISTER_RAMDISK) ( 
    IN UINT64 RamDiskBase,
    IN UINT64 RamDiskSize,
    IN EFI_GUID *RamDiskType,
    IN EFI_DEVICE_PATH *ParentDevicePath OPTIONAL,
    OUT EFI_DEVICE_PATH_PROTOCOL **DevicePath 
    );

Parameters

RamDiskBase  The base address of registered RAM disk.
RamDiskSize  The size of registered RAM disk.
RamDiskType  The type of registered RAM disk. The GUID can be any of the values defined in Section 10.3.5.9, or a vendor defined GUID.
ParentDevicePath  Pointer to the parent device path. If there is no parent device path then ParentDevicePath is NULL.
DevicePath  On return, points to a pointer to the device path of the RAM disk device. If ParentDevicePath is not NULL, the returned DevicePath is created by appending a RAM disk node to the parent device path. If ParentDevicePath is NULL, the returned DevicePath is a RAM disk device path without appending. This function is responsible for allocating the buffer DevicePath with the boot service AllocatePool().

Description

The Register function is used to register a specified RAM Disk. The consumer of this API is responsible for allocating the space of the RAM disk and deciding the initial content of the RAM disk. The producer of this API is responsible for installing the RAM disk device path and block I/O related protocols on the RAM disk device handle.

RamDiskBase, RamDiskSize and RamDiskType are used to fill RAM disk device node. If RamDiskSize is 0, then EFI_INVALID_PARAMETER is returned. If RamDiskType is NULL, then EFI_INVALID_PARAMETER is returned.

DevicePath returns the device path of the registered RAM disk. If DevicePath is NULL, then EFI_INVALID_PARAMETER is returned. If there are not enough resources to allocate the device path node, then EFI_OUT_OF_RESOURCES is returned. Otherwise, DevicePath is allocated with the boot service AllocatePool(). If ParentDevicePath is not NULL the DevicePath instance is created by appending a RAM disk device node to the ParentDevicePath. If ParentDevicePath is NULL the DevicePath instance is a pure RAM disk device path. If the created DevicePath instance is already present in the handle database, then EFI_ALREADY_STARTED is returned.
**Status Codes Returned**

<table>
<thead>
<tr>
<th>Code</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>EFI_SUCCESS</td>
<td>The RAM disk is registered successfully.</td>
</tr>
<tr>
<td>EFI_INVALID_PARAMETER</td>
<td>DevicePath or RamDiskType is NULL.</td>
</tr>
<tr>
<td>EFI_INVALID_PARAMETER</td>
<td>RamDiskSize is 0.</td>
</tr>
<tr>
<td>EFI_ALREADY_STARTED</td>
<td>A Device Path Protocol instance to be created is already present in the handle database.</td>
</tr>
<tr>
<td>EFI_OUT_OF_RESOURCES</td>
<td>The RAM disk register operation fails due to resource limitation.</td>
</tr>
</tbody>
</table>

**EFI_RAM_DISK_PROTOCOL.Unregister()**

**Summary**

This function is used to unregister a RAM disk specified by DevicePath.

**Prototype**

```c
typedef EFI_STATUS
    (EFIAPIC *EFI_RAM_DISK_UNREGISTER_RAMDISK) (IN EFI_DEVICE_PATH_PROTOCOL *DevicePath);
```

**Parameters**

- **DevicePath**
  A pointer to the device path that describes a RAM Disk device.

**Description**

The Unregister function is used to unregister a specified RAM Disk. The producer of this protocol is responsible for uninstalling the RAM disk device path and block I/O related protocols and freeing the RAM disk device handle. It is the consumer of this protocol’s responsibility to free the memory used by this RAM disk.
Status Codes Returned

<table>
<thead>
<tr>
<th>Code</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>EFI_SUCCESS</td>
<td>The RAM disk is unregistered successfully.</td>
</tr>
<tr>
<td>EFI_INVALID_PARAMETER</td>
<td>DevicePath is NULL.</td>
</tr>
<tr>
<td>EFI_UNSUPPORTED</td>
<td>The device specified by DevicePath is not a valid ramdisk device path and not supported by the driver.</td>
</tr>
<tr>
<td>EFI_NOT_FOUND</td>
<td>The RAM disk pointed by DevicePath doesn’t exist.</td>
</tr>
</tbody>
</table>

13.18 Partition Information Protocol

Summary

Installed along with EFI_BLOCK_IO_PROTOCOL for logical partitions. The PARTITION_INFORMATION_PROTOCOL provides cached partition information for MBR and GPT partition types.

Set System to 1 for partition identified as EFI_SYSTEM_PARTITIONS, otherwise set System to 0.

Set Type to PARTITION_TYPE_OTHER for partitions that are not GPT or MBR to indicate no cached data.

GUID

```c
#define EFI_PARTITION_INFO_PROTOCOL_GUID \
{ \n    0x8cf2f62c, 0xbc9b, 0x4821, {0x80, 0x8d, 0xec, 0x9e, \n        0xc4, 0x21, 0xa1, 0xa0} \n}
```

Protocol Interface Structure
#define EFI_PARTITION_INFO_PROTOCOL_REVISION 0x0001000
#define PARTITION_TYPE_OTHER  0x00
#define PARTITION_TYPE_MBR   0x01
#define PARTITION_TYPE_GPT   0x02

#pragma pack (1)

typedef struct {
    UINT32         Revision;
    UINT32         Type;
    UINT8         System;
    UINT8         Reserved[7];
    union {
        ///
        /// MBR data
        ///
        MBR_PARTITION_RECORD Mbr;
    }
    ///
    /// GPT data
    ///
    EFI_PARTITION_ENTRY Gpt;
} Info;
} EFI_PARTITION_INFO_PROTOCOL;

#pragma pack ()

See Section 5.2.1 for the definition of MBR_PARTITION_RECORD.

See Section 5.3.3 for the definition of EFI_PARTITION_ENTRY.

Parameters

Revision
Set to EFI_PARTITION_INFO_PROTOCOL_REVISION

Type
Partition info type (MBR, GPT, or Other).

System
If 1, partition describes an EFI System Partition.

Mbr
MBR information, if type is MBR.

Gpt
GPT information, if type is GPT

Description

The EFI_PARTITION_INFO_PROTOCOL is a simple protocol used to cache the partition information for potential File System Drivers.

Care must be taken by UEFI utilities that manipulate partitions. The utility must gain exclusive access to the physical disk to cause the partition driver to be stopped before it changes the partition information. If the exclusive request is not granted, then the utility must reset the system after changing the partition information.

When Type is set to PARTITION_TYPE_OTHER, data in the union Info is undefined.
13.19 NVDIMM Label Protocol

EFI_NVDIMM_LABEL_PROTOCOL
This section provides a detailed description of the EFI_NVDIMM_LABEL_PROTOCOL. For a high-level overview of the NVDIMM Label protocol see the Label Storage Area Description section.

Summary
Provides services that allow management of labels contained in a Label Storage Area that are associated with a specific NVDIMM Device Path. The labels describe how the data on the NVDIMM is organized in to namespaces, the layout being utilized, logical block size, unique label identifier, label state, etc.

GUID
#define EFI_NVDIMM_LABEL_PROTOCOL_GUID {
  0xd40b6b80,0x97d5,0x4282, \ 
  0xbb,0x1d,0x22,0x3a,0x16,0x91,0x80,0x58}

Protocol Interface Structure
typedef struct _EFI_NVDIMM_LABEL_PROTOCOL {
  EFI_NVDIMM_LABEL_STORAGE_INFORMATION LabelStorageInformation;
  EFI_NVDIMM_LABEL_STORAGE_READ   LabelStorageRead;
  EFI_NVDIMM_LABEL_STORAGE_WRITE    LabelStorageWrite;
} Parameters

LabelStorageInformation Reports the size of the Label Storage Area and the maximum amount of label data that can be read in a single call to LabelStorageRead or written in a single call to LabelStorageWrite.

LabelStorageRead Returns the label data stored for the NVDIMM at the requested offset and length in the Label Storage Area.

LabelStorageWrite Writes the label data stored for the NVDIMM at the requested offset and length in the Label Storage Area.

EFI_NVDIMM_LABEL_PROTOCOL.LabelStorageInformation()
Summary
Retrieves the Label Storage Area size and the maximum transfer size for the LabelStorageRead and LabelStorageWrite methods that are associated with a specific NVDIMM Device Path.

Prototype
typedef
EFI_STATUS
(EIFIAPI *EFI_NVDIMM_LABEL_STORAGE_INFORMATION) ( 
  IN EFI_NVDIMM_LABEL_PROTOCOL *This, 
  OUT UINT32 *SizeOfLabelStorageArea,
OUT UINT32 *MaxTransferLength

Parameters

This

A pointer to the EFI_NVDIMM_LABEL_PROTOCOL instance.

SizeOfLabelStorageArea

The size of the Label Storage Area for the NVDIMM in bytes.

MaxTransferLength

The maximum number of bytes that can be transferred in a single call to LabelStorageRead or LabelStorageWrite.

Description

Retrieves the Label Storage Area size and the maximum transfer size for the LabelStorageRead and LabelStorageWrite methods.

Status Codes Returned

<table>
<thead>
<tr>
<th>Status Code</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>EFI_SUCCESS</td>
<td>The size of the Label Storage Area and maximum transfer size returned are valid.</td>
</tr>
<tr>
<td>EFI_ACCESS_DENIED</td>
<td>The Label Storage Area for the NVDIMM device is not currently accessible</td>
</tr>
<tr>
<td>EFI_DEVICE_ERROR</td>
<td>A physical device error occurred and the data transfer failed to complete</td>
</tr>
</tbody>
</table>

EFI_NVDIMM_LABEL_PROTOCOL.LabelStorageRead()

Summary

Retrieves label data for the NVDIMM for the requested byte offset and length from within the Label Storage Area that are associated with a specific NVDIMM Device Path.

Prototype

typedef

EFI_STATUS

(EIFIAPI *EFI_NVDIMM_LABEL_STORAGE_READ) (  
    IN CONST EFI_NVDIMM_LABEL_PROTOCOL *This,
    IN UINT32 Offset,
    IN UINT32 TransferLength,
    OUT UINT8 *LabelData

);

Parameters

This

A pointer to the EFI_NVDIMM_LABEL_PROTOCOL instance.

Offset

The byte offset within the Label Storage Area to read from.

TransferLength

Number of bytes to read from the Label Storage Area beginning at the byte Offset specified. A TransferLength of 0 reads no data.

LabelData

The return label data read at the requested offset and length from within the Label Storage Area.
Description
Retrieves the label data for the requested offset and length from within the Label Storage Area for the NVDIMM. See the Label Index Block and Label Definitions sections below for details on the contents of the label data.

Status Codes Returned

<table>
<thead>
<tr>
<th>Code</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>EFI_SUCCESS</td>
<td>The label data from the Label Storage Area for the NVDIMM was read successfully at the specified Offset and TransferLength and LabelData contains valid data.</td>
</tr>
</tbody>
</table>
| EFI_INVALID_PARAMETER | Returned if any of the following are true:  
- Offset > SizeOfLabelStorageArea reported in the LabelStorageInformation return data.  
- Offset + TransferLength is > SizeOfLabelStorageArea reported in the LabelStorageInformation return data.  
- TransferLength is > MaxTransferLength reported in the LabelStorageInformation return data. |
| EFI_ACCESS_DENIED   | The Label Storage Area for the NVDIMM device is not currently accessible and labels cannot be read at this time.                                |
| EFI_DEVICE_ERROR    | A physical device error occurred and the data transfer failed to complete.                                                                   |

EFI_NVDIMM_LABEL_PROTOCOL.LabelStorageWrite()

Summary
Writes label data for the NVDIMM for the requested byte offset and length to the Label Storage Area that are associated with a specific NVDIMM Device Path.

Prototype

typedef
EFI_STATUS
(EIFIAPI *EFI_NVDIMM_LABEL_STORAGE_WRITE) (  
    IN CONSTEFI_NVDIMM_LABEL_PROTOCOL *This,  
    IN UINT32 Offset,  
    IN UINT32 TransferLength,  
    IN UINT8 *LabelData)
);

Parameters

This  
A pointer to the EFI_NVDIMM_LABEL_PROTOCOL instance.

Offset  
The byte offset within the Label Storage Area to write to.

TransferLength  
Number of bytes to write to the Label Storage Area beginning at the byte Offset specified. A TransferLength of 0 writes no data.
LabelBuffer

The label data to write at the requested offset and length from within the Label Storage Area.

Description

Writes the label data for the requested offset and length in to the Label Storage Area for the NVDIMM. See the Label Index Block and Label Definitions sections below for details on the contents of the label data.

Status Codes Returned

<table>
<thead>
<tr>
<th>Status Code</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>EFI_SUCCESS</td>
<td>The LabelData for the Label Storage Area for the NVDIMM was written successfully at the specified Offset and TransferLength.</td>
</tr>
<tr>
<td>EFI_INVALID_PARAMETER</td>
<td>Returned this status if any of the following are true: - Offset &gt; SizeOfLabelStorageArea reported in the LabelStorageInformation return data. - Offset + LabelBufferLength is &gt; SizeOfLabelStorageArea reported in the LabelStorageInformation return data. - TransferLength is &gt; MaxTransferLength reported in the LabelStorageInformation return data.</td>
</tr>
<tr>
<td>EFI_ACCESS_DENIED</td>
<td>The Label Storage Area for the NVDIMM device is not currently accessible and labels cannot be written at this time.</td>
</tr>
<tr>
<td>EFI_DEVICE_ERROR</td>
<td>A physical device error occurred and the data transfer failed to complete</td>
</tr>
</tbody>
</table>

Label Index Block Definitions

```c
#define EFI_NVDIMM_LABEL_INDEX_SIG_LEN 16
#define EFI_NVDIMM_LABEL_INDEX_ALIGN 256

typedef struct EFI_NVDIMM_LABEL_INDEX_BLOCK {
    char8 Sig[EFI_NVDIMM_LABEL_INDEX_SIG_LEN];
    uint8 Flags[3];
    uint8 LabelSize;
    uint32 Seq;
    uint64 MyOff;
    uint64 MySize;
    uint64 OtherOff;
    uint64 LabelOff;
    uint32 NSlot;
    uint16 Major;
    uint16 Minor;
    uint64 Checksum;
    uint8 Free[];
} LABEL_INDEX_BLOCK;
```

Sig

Signature of the Index Block data structure. Must be “NAMESPACE_INDEX\0”.

Flags
Boolean attributes of this Label Storage Area. There are no flag bits defined at this time, so this field must be zero.

**LabelSize**

*Size of each label in bytes, 128 bytes << LabelSize. 1 means 256 bytes, 2 means 512 bytes, etc. Shall be 1 or greater.*

**Seq**

Sequence number used to identify which of the two Index Blocks is current. Only the least-significant two bits of this field are used in the current definition, rotating through the values depicted in Figure 2: Cyclic Sequence Numbers in Label Index Block below. The other bits must be zero.

![Figure 13-2 Cyclic Sequence Numbers in Label Index Block](image)

Each time an Index Block is written, the sequence number of the current Index Block is “incremented” by moving to the next value clockwise as shown.

Since there are two Index Blocks, written alternatively with successive sequence numbers, the older Index Block’s sequence number will be immediately behind (counter-clockwise to) the current Index Block’s sequence number. This property is used during software initialization to identify the current Index Block.

The sequence number 00 is used to indicate an uninitialized or invalid Index Block. Software never writes the sequence number 00, so a correctly check-summed Index Block with this sequence number probably indicates a critical error. When software discovers this case it treats it as an invalid Index Block indication. If two Index Blocks with identical sequence numbers are found, software shall treat the Index Block at the higher offset as the valid Index Block.

**MyOff**

The offset of this Index Block in the Label Storage Area.

**MySize**

The size of this Index Block in bytes. This field must be a multiple of the EFI_NVDIMM_LABEL_INDEX_ALIGN.

**OtherOff**

The offset of the other Index Block paired with this one.
LabelOff
The offset of the first slot where labels are stored in this Label Storage Area.

NSlot
The total number of slots for storing labels in this Label Storage Area. The NSlot field is typically calculated once at Label Storage Area initialization as described in the Initial Label Storage Area Configuration description.

Major
Major version number. Value shall be 1.

Minor
Minor version number. Value shall be 2.

Checksum
64-bit Fletcher64 checksum of all fields in this Index Block. This field is considered zero when the checksum is computed. For references to the Fletcher64 algorithm see “Links to UEFI-Related Documents” (http://uefi.org/uefi) under the heading “Fletcher64 Checksum Algorithm”

Free
Array of unsigned bytes implementing a bitmask that tracks which label slots are free. A bit value of 0 indicates in use, 1 indicates free. The size of this field is the number of bytes required to hold the bitmask with NSlot bits, padded with additional zero bytes to make the Index Block size a multiple of EFI_NVDIMM_LABEL_INDEX_ALIGN. Any bits allocated beyond NSlot bits must be zero.

The bitmask is organized where the label slot with the lowest offset in the Label Storage Area is tracked by the least significant bit of the first byte of the free array. Missing from the above layout is a total count of free slots. Since the common use case for the Label Storage Area is to read all labels during software initialization, it is recommended that software create a total free count (or in use count, or both), maintained at run-time. Rules for maintaining the Index Blocks are described in the Label Rules Description and Validating Index Blocks Description below. See the Initial Label Storage Area Configuration section for a more details on how the total number of slots are calculated.
Label Definitions

#define EFI_NVDIMM_LABEL_NAME_LEN 64

// Constants for Flags field
#define EFI_NVDIMM_LABEL_FLAGS_ROLABEL 0x00000001
#define EFI_NVDIMM_LABEL_FLAGS_LOCAL 0x00000002
#define EFI_NVDIMM_LABEL_FLAGS_SPACOOKIE_BOUND 0x00000010

// This reserved flag is utilized on older implementations
// and has been deprecated. Do not use
#define EFI_NVDIMM_LABEL_FLAGS_RESERVED 0x00000004
#define EFI_NVDIMM_LABEL_FLAGS_UPDATING 0x00000008

typedef struct EFI_NVDIMM_LABEL{
  EFI_GUID   Uuid;
  CHAR8      Name[EFI_NVDIMM_LABEL_NAME_LEN];
  UINT32     Flags;
  UINT16     NLabel;
  UINT16     Position;
  UINT64     SetCookie;
  UINT64     LbaSize;
  UINT64     Dpa;
  UINT64     RawSize;
  UINT32     Slot;
  UINT8      Alignment;
  UINT8      Reserved[3];
  EFI_GUID   TypeGuid;
  EFI_GUID   AddressAbstractionGuid;
  UINT64     SPALocationCookie;
  UINT8      Reserved1[80];
  UINT64     Checksum;
} EFI_NVDIMM_LABEL;

Uuid

Unique Label Identifier UUID per RFC 4122. This field provides two functions. First, the namespace is associated with a UUID that software can use to uniquely identify it and providing a way for the namespace to be matched up with applications using it. Second, when multiple labels are required to describe a namespace, the UUID is the mechanism used to group the labels together. See the additional descriptions below describing the process for grouping the labels together by UUID, checking for missing labels, recovering from partial label changes, etc.

Name

NULL-terminated string using UTF-8 character formatting. The Name field is optionally used by software to store a more friendly name for the namespace. When this field is unused, it contains zeros.

If there is a name for a Local Namespace, as indicated by the EFI_NVDIMM_LABEL_FLAGS_LOCAL Flags, the name shall be stored in the first label of the set. All Name fields in subsequent labels for that Local Namespace are ignored.
The **Name** field can be set at label creation time, or updated by following the rules in the additional descriptions below.

**Flags**

Boolean attributes of this namespace. See the additional description below on the use of the flags. The following values are defined:

- **EFI_NVDIMM_LABEL_FLAGS_ROLABEL** – The label is read-only. This indicates the namespace is exported to a domain where configuration changes to the label are not allowed, such as a virtual machine. This indicates that device software and manageability software should refuse to make changes to the labels. This is not a security mechanism, but a usability feature instead. In cases where **EFI_NVDIMM_LABEL_FLAGS_ROLABEL** is set, such as virtual machine guests, attempting to make configuration changes that affect the labels will fail (i.e. because the VM guest is not in a position to make the change correctly). For these cases, the VMM can set the **EFI_NVDIMM_LABEL_FLAGS_ROLABEL** bit on the label exposed to the guest to provide a better user experience where manageability refuses to make changes with a friendlier error message.

- **EFI_NVDIMM_LABEL_FLAGS_LOCAL** – When set, the complete label set is local to a single NVDIMM Label Storage Area. When clear, the complete label set is contained on multiple NVDIMM Label Storage Areas. If NLabel is 1, then setting this flag is optional and it is implied that the **EFI_NVDIMM_LABEL_FLAGS_LOCAL** flag is set, as the complete label set is local to a single NVDIMM Label Storage Area.

- **EFI_NVDIMM_LABEL_FLAGS_UPDATING** – When set, the label set is being updated. During an operation that may require updating multiple Label Storage Areas, the **EFI_NVDIMM_LABEL_FLAGS_UPDATING** flag is used to make the update atomic across interruptions. Updates happen in two phases, first writing the label with the **EFI_NVDIMM_LABEL_FLAGS_UPDATING** flag set, second writing the updated label without the **EFI_NVDIMM_LABEL_FLAGS_UPDATING** flag. As described in Recovery Steps for a Non-Local Label Set Description, this allows recovery actions during software initialization to either roll back or roll forward the multiple Label Storage Area changes. If **EFI_NVDIMM_LABEL_FLAGS_LOCAL** is set, the labels are contained in a single Label Storage Area and there is no need to set **EFI_NVDIMM_LABEL_FLAGS_UPDATING**, since the label can be written in one atomic operation.

- **EFI_NVDIMM_LABEL_FLAGS_SPACOOKIE_BOUND** – When set, the SPA Location Cookie in the namespace label is valid and should match the current value in the NFIT SPA Range Structure.

**NLabel**

Total number of labels describing this namespace. The **NLabel** field contains the number of labels required to describe a namespace.

**Position**

Position of this label in list of labels for this namespace. See **NLabel** description above. In the non-local case, each label is numbered as to its position in the list of labels using the **Position** field. For example, the common case where a namespace requires exactly one label, **NLabel** will be 1 and **Position** will be 0. If a namespace is built on an Interleave Set that spans multiple Label Storage Areas, each Label Storage Area will contain a label with increasing **Position** values to show each labels position in the set. For Local Namespaces, **NLabel** is valid only for the first label (lowest DPA) and position shall be 0 for that label. As part of organizing and validating the labels, SW shall have organized the labels from lowest to highest DPA so the first label in that ordered list of labels will have the lowest DPA. **Position** and **NLabel** for all
subsequent labels in that namespace shall be set to 0xFF. See the Local Namespace description in the Validating Labels Description section for details.

**SetCookie**

Interleave Sets and the NVDIMMs they contain are defined in the NFIT and the Uuid in the label is used to uniquely identify each interleave set. The SetCookie is utilized by SW to perform consistency checks on the Interleave Set to verify the current physical device configuration matches the original physical configuration when the labels were created for the set. The label is considered invalid if the actual label set cookie doesn’t match the cookie stored here. The SetCookie field in each label for that namespace is derived from data in the NVDIMM’s Serial Presence Detect (SPD). See the SetCookie Description section below for SetCookie details. For references to the JEDEC SPD annex see “Links to UEFI-Related Documents” (http://uefi.org/uefi) under the heading "JEDEC SPD Annex”

**LbaSize**

This is the default logical block size in bytes and may be superseded by a block size that is specified in the AbstractionGuid.
- A non-zero value indicates the logical block size that is being emulated.
- A value of zero indicates an unspecified size and its meaning is implementation specific

**Dpa**

The DPA is the Device Physical Address where the NVM contributing to this namespace begins on this NVDIMM.

**RawSize**

The extent of the DPA contributed by this label.

**Slot**

Current slot in the Label Storage Area where this label is stored.

**Alignment**

Alignment hint used to advertise the preferred alignment of the data from within the namespace defined by this label.

**Reserved**

Shall be 0

**TypeGuid**

Range Type GUID that describes the access mechanism for the specified DPA range. The GUIDs utilized for the type are defined in the ACPI 6.0 specification in the NVDIMM FW Interface Table (NFIT) chapter. Those values are utilized here to describe the Type of namespace the label is describing. See the Address Range Type GUID field described in the System Physical Address (SPA) Range Structure table.

**AddressAbstractionGuid**

Identifies the address abstraction mechanism for this namespace. A value of 0 indicates no mechanism used.
SPALocationCookie
When creating the label, this value is set to the value from the NFIT SPA Range Structure if the SPALocationCookie flag (bit 2) is set. If EFI_NVDIMM_LABEL_FLAGS_SPACOOKIE_BOUND is set, the SPALocationCookie value stored in the namespace label should match the current value in the NFIT SPA Range Structure. Otherwise, the data may not be read correctly.

Reserved1
Shall be 0

Checksum
64-bit Fletcher64 checksum of all fields in this Label. This field is considered zero when the checksum is computed. For references to the Fletcher64 algorithm see “Links to UEFI-Related Documents” (http://uefi.org/uefi) under the heading “Fletcher64 Checksum Algorithm”

SetCookie Definition
typedef struct EFI_NVDIMM_LABEL_SET_COOKIE_INFO {
    typedef struct EFI_NVDIMM_LABEL_SET_COOKIE_MAP {
        UINT64 RegionOffset;
        UINT32 SerialNumber;
        UINT16 VendorId;
        UINT16 ManufacturingDate;
        UINT8 ManufacturingLocation;
        UINT8 Reserved[31];
    } Mapping[NumberOfNvdimmsInInterleaveSet];
};

NumberOfNvdimmsInInterleaveSet
The number of NVDIMMs in the interleave set. This is 1 if EFI_NVDIMM_LABEL_FLAGS_LOCAL Flags is set indicating a Local Namespaces.

RegionOffset
The Region Offset field from the ACPI NFIT NVDIMM Region Mapping Structure for a given entry. This determines the entry’s position in the set. Region offset is 0 for Local Namespaces.

SerialNumber
The serial number of the NVDIMM, assigned by the module vendor. This field shall be set to the value of the NVDIMM Serial Presence Detect (SPD) Module Serial Number field defined by JEDEC with byte 0 set to SPD byte 325, byte 1 set to SPD byte 326, byte 2 set to SPD byte 327, and byte 3 set to SPD byte 328. For references to the JEDEC SPD annex see “Links to UEFI-Related Documents” (http://uefi.org/uefi) under the heading “JEDEC SPD Annex”

VendorId
The identifier indicating the vendor of the NVDIMM. This field shall be set to the value of the NVDIMM SPD Module Manufacturer ID Code field with byte 0 set to DDR4 SPD byte 320 and byte 1 set to DDR4
SPD byte 321. For references to the JEDEC SPD annex see “Links to UEFI-Related Documents” (http://uefi.org/uefi) under the heading “JEDEC SPD Annex”

**ManufacturingDate**

The manufacturing date of the NVDIMM, assigned by the module vendor. This field shall be set to the value of the NVDIMM SPD Module Manufacturing Date field with byte 0 set to SPD byte 323 and byte 1 set to SPD byte 324. For references to the JEDEC SPD annex see “Links to UEFI-Related Documents” (http://uefi.org/uefi) under the heading “JEDEC SPD Annex”

**ManufacturingLocation**

The manufacturing location from for the NVDIMM, assigned by the module vendor. This field shall be set to the value of the NVDIMM SPD Module Manufacturing Location field (SPD byte 322). For references to the JEDEC SPD annex see “Links to UEFI-Related Documents” (http://uefi.org/uefi) under the heading “JEDEC SPD Annex”

**Reserved**

Shall be 0

**SetCookie Description**

This value is used to detect a change in the set configuration that has rendered existing data invalid and otherwise validates that the namespace belongs to a given NVDIMM. For each set create a data structure of the form EFI_NVDIMM_LABEL_SET_COOKIE_INFO. The SetCookie is then calculated by sorting the Mapping[] array by RegionOffset and then taking the Fletcher64 sum of the total EFI_NVDIMM_LABEL_SET_COOKIE_INFO structure. For references to the Fletcher64 algorithm see “Links to UEFI-Related Documents” (http://uefi.org/uefi) under the heading “Fletcher64 Checksum Algorithm”

**Label Storage Area Description**

Namespaces are defined by Labels which are stored in the Label Storage Area(s) and accessed via means described in the Label Rules Description.

The figure below shows the organization of the Label Storage Area. A header called the Index Block appears twice at the top of the Label Storage Area. This provides a powerfail-safe method for updating the information in the Label Storage Area by alternating between the two Index Blocks when writing (more details on this mechanism below).
Following the Index Blocks, an array for storing labels takes up the remainder of the Label Storage Area. The size of the Label Storage Area is NVDIMM implementation specific. The Index Blocks contain a bitmap which indicates which label slots are currently free and which are in use. The same powerfail-safe mechanism used for updating the Index Blocks covers the update of labels in the Label Storage Area.

The powerfail-safe update mechanism uses the principle of avoiding writes to active metadata. Instead, a shadow copy is updated and checksums and sequence numbers are used to make the last written copy active (a complete description of this mechanism is in Updating an Existing Label Description).

### Initial Label Storage Area Configuration

The size of an Index Block depends on how many label slots fit into the Label Storage Area. The minimum size of an Index Block is 256 bytes and the size must be a multiple of EFI_NVDIMM_LABEL_INDEX_ALIGN bytes. As necessary, padding with zero bytes at the end of the structure is used to meet these size requirements. The minimum size of the Label Storage Area is large enough to hold 2 index blocks and 2 labels. As an example, for Label Storage Areas of 128KB and 256KB, the corresponding Index Block size is 256 or 512 bytes:

<table>
<thead>
<tr>
<th>Example: &lt;= 256 bytes</th>
<th>Example: &gt; 256 bytes</th>
</tr>
</thead>
<tbody>
<tr>
<td>Size of the Index Block field up to the free field</td>
<td>72 bytes</td>
</tr>
<tr>
<td>Bytes required for a bitmask of 1024 labels (the number of labels that fit into a 128KB Label Storage Area)</td>
<td>128 bytes</td>
</tr>
<tr>
<td>Padding to meet next increment of 256 bytes</td>
<td>56 bytes</td>
</tr>
<tr>
<td>Total size of the Index Block</td>
<td>256 bytes</td>
</tr>
</tbody>
</table>
Before Index Blocks and labels can be utilized, the software managing the Label Storage Area must determine the total number of labels that will be supported and utilizing the description above, calculate the size of the Index Blocks required. Once the initial Label Storage Area is written with the first Index Blocks (typically done when the first Label needs to be written), the total number of slots is fixed and this initial calculation is not performed again.

**Label Description**

Each slot in the Label Storage Area is either free or contains an active label.

In the cases where multiple labels are used to describe a namespace, the label fields `NLabel` and `Position` provide an ordering (“label one of two, label two of two”) so that incomplete label sets can be detected.

A namespace is described by one or more labels. Local namespaces describe one or more device physical address ranges from a single NVDIMM while non-Local namespaces describe a single SPA range that may have contributions from 2 or more NVDIMMs. The number of labels needed to describe a non-Local namespace is equal to the number of NVDIMMs contributing to the SPA range, 1 per-NVDIMM. For a Local namespace any number, up to the max number of labels supported by the Index Block / Label Storage Area, of device physical address ranges in the given NVDIMM can be described.

**Label Rules Description**

All the algorithms related to labels in this specification assume single-threaded / non-reentrant execution. The algorithm for updating labels guarantees that at least one slot in the Label Storage Area will be free, ensuring it is always possible to update labels using this method.

Software shall maintain the following invariants to use the on-media data structures correctly and to inter-operate with other software components.

At all times, the following must be true:

- The size of the Label Storage Area is known (this must be true even if no namespace metadata has been written yet). The Label Storage Area size is queried from the NVDIMM.
- The Label Storage Area either contains no valid Index Blocks, indicating there are no labels on the NVDIMM (all slots free), or the validation rules below produce a single, valid, Index Block.
- 2 free slots are required in order to add a Label. Having only a single free slot indicates that no more labels can be added. Only fully written, active labels, and full-written labels with the `EFI_NVDIMM_LABEL_FLAGS_UPDATING` flag are marked in-use by the Index Block.
- Write to active label slots are not allowed; all updates to labels must be done by writing to free slots and then updating the Index Block to make them active.

**Validating Index Blocks Description**

The following tests shall pass before software considers Index Blocks valid:

- Both Index Blocks must be read successfully from the Label Storage Area.
- Any Index Block with an incorrect `Sig` field is invalid.
- Any Index Block with an incorrect `Checksum` is invalid.
- Any Index Block with an incorrect `MyOff`, `MySize`, or `OtherOff` field is invalid.
- Any Index Block with a sequence number `Seq` of zero is invalid.
• If two valid Index Blocks remain, after passing all the above tests, and their sequence numbers match, the Index Block at the lower offset in the Label Storage Area is invalid.

• If two Index Blocks remain, after passing all the above tests, their sequence numbers are compared and the block whose sequence number is the predecessor of the other (immediately counter-clockwise to it, as shown in Figure Z: Cyclic Sequence Numbers in Label Index Block in the Seq description) is invalid.

• If one Index Block remains, that is the current, valid block and software should make note that the next update to the Index Block will write the other Index Block. However, if no valid Index Blocks remain, all slots are considered free and the next update to the index will write to the lower-addressed block location (i.e. the start of the Label Storage Area).

Validating Labels Description
The following tests shall pass before software considers individual Labels slots valid:

• The corresponding free bit for the label Slot in the Index Block Free array must be clear (i.e. label slot is active).

• The label Checksum shall validate.

• The Slot value in the Label shall match the logical slot location of the Label.

• The SetCookie field in the label matches the expected value as described in SetCookie Definition.

• The address range type GUID TypeGuid shall match the System Physical Address Range Structure that describes the access mechanism for this namespace. For Hardware Block Namespaces it shall match the GUID for the NVDIMM Block Data Window Region.

For Local Namespaces:

• If 2 or more labels share the same Dpa value, all labels with the duplicated value are considered invalid.

• The count of all valid labels for a given namespace Uuid shall match the NLabel value in the first label.

• The first label, the label with the lowest Dpa value, shall have Position 0 and non-zero NLabel value.

• All labels other than the first have Position and NLabel set to 0xff.

Reading Labels Description
For a given NVDIMM, the following steps are used to read one or more labels for validation and namespace assembly:

Pre-condition: both Index Blocks have been read and the rules in Validating Index Blocks Description have been followed to determine the current valid Index Block.

• Check that the label at a given slot is active. Specifically bit N is clear in the Free bitmask field where N corresponds to the logical slot number label.

• Read the label in that slot at the offset given by:

\[
(2 \times \text{sizeof(EFI_NVDIMM_LABEL_INDEX_BLOCK)} + \text{slot} \times \text{sizeof(EFI_NVDIMM_LABEL)})
\]
Recovery Steps for a Non-Local Label Set Description

Given that a non-Local Label set potentially spans multiple Label Storage Areas for multiple NVDIMMs it is not possible to guarantee that the set is updated atomically with respect to unexpected system interruption. Recovery shall be performed before validating the set to roll the set forward to a consistent state or invalidate / free the label slots corresponding to an inconsistent state. Note that individual Index Block updates are safe with respect to unexpected system interruption given the sequence number mechanism for indicating the currently active Index Block.

The sequence below describes how the `EFI_NVDIMM_LABEL_FLAGS_UPDATING` flag is used when validating a non-Local Label Set.

- **Pre-condition:** The labels have been read.
- For each set of labels with the same UUID, if no labels in the set are found with the `EFI_NVDIMM_LABEL_FLAGS_UPDATING` flag set, then no recovery is required for that set.
- For the sets where `EFI_NVDIMM_LABEL_FLAGS_UPDATING` appear at least once, if the set is incomplete (some NVDIMMs in the set do not contain a label in the Label Storage Area with the UUID), the recovery action is to roll back the interrupted create operation that left this state. I.e. for each NVDIMM in the set containing a label with the given UUID, delete the label.
- For a set where `EFI_NVDIMM_LABEL_FLAGS_UPDATING` appears at least once and the set is otherwise complete (each NVDIMM in the Interleave Set contains a label with the UUID, some with `EFI_NVDIMM_LABEL_FLAGS_UPDATING` set, some with `EFI_NVDIMM_LABEL_FLAGS_UPDATING` clear), the recovery action is to roll forward the change that was interrupted. I.e. for each NVDIMM in the set if `EFI_NVDIMM_LABEL_FLAGS_UPDATING` is set, write an updated label with `EFI_NVDIMM_LABEL_FLAGS_UPDATING` clear and with the name field copied from the first label in the set (the label with a `Position` field of 0).

Recovery Steps for a Local Label Set Description

Given that a Local Label set is always contained in a single Label Storage Area for a single NVDIMM, labels are added/updated atomically, as long as there is a free Label available as outlined in Label Storage Area Description and Label Description. `EFI_NVDIMM_LABEL_FLAGS_UPDATING` should not be set for Local sets and no additional recovery is required.

Assembling Labels into Complete Sets Description

After collecting a set of labels corresponding to a given UUID and performing the recovery actions on the set, software shall follow the steps in this section to assemble complete sets of labels representing usable namespaces:

1. **Precondition:** Labels have been read and the recovery actions have been taken.
2. For each set of labels with the same `Uuid`
   a. If the set describes a non-Local namespace, it is considered complete if labels with unique `Position` fields are found for every position from 0 to `NLabel - 1`.
   b. If the set describes a Local namespace, it is considered complete if a valid first label is found, according to the validation rules, and the number of labels in the set matches...
c NLabel.
The recovery action for the case where software finds incomplete namespaces is implementation specific.

Updating an Existing Label Description

Updating an existing label in the Label Storage Area requires the software to follow these steps:

1. Pre-conditions: the software has an updated label constructed to be written to a specific NVDIMM's Label Storage Area. There is at least 1 free slot in the Label Storage Area Free bitmask.
2. The software chooses a free slot from the Index Block, fills in that slot number in the label’s Slot field
3. The software writes the updated label to that slot in the Label Storage Area
4. The software updates the Index Block by taking the current Index Block, setting the appropriate bit in the Free field to make the old version of the label inactive and clearing the appropriate bit in the Free field to make the new version active, incrementing the sequence number as shown in Figure Z: Cyclic Sequence Numbers in Label Index Block in the Seq description, and then writing the Index Block over the inactive Index Block location (making this location the new active Index Block if the write succeeds)

Deleting a Label Description

The software updates the Index Block by taking the current active Index Block, setting the appropriate bit in the Free field to make the deleted label inactive, incrementing the sequence number as shown in Figure Z: Cyclic Sequence Numbers in Label Index Block in the Seq description, and then writing the new Index Block over the inactive Index Block location (making this location the new active Index Block if the write succeeds)

Creating Namespaces Description

Namespace creation procedures are different for Local vs non-Local namespaces. A Local namespace is created from 1 or more DPA ranges of a single NVDIMM, while a non-Local namespace is created from a single range contributed from multiple NVDIMMs. Both procedures share a common flow for establishing new labels in an Index Block.

Writing New Labels Description

Transitioning a label slot from free to active shall follow this sequence:

1. Pre-conditions: the software has a new label constructed to be written to a specific NVDIMM’s Label Storage Area. Because of the free Label rules outlined in Label Storage Area Description and Label Description, there are at least 2 free slots in the Label Storage Area as described in the Label Rules Description and Label Description sections. Choose a free slot from the Index Block, fills in that slot number in the label’s Slot field
2. Write the new label to that slot in the Label Storage Area
3. Update the Index Block by taking the current Index Block, clearing the appropriate bit in the Free field, incrementing the sequence number as shown in Figure Z: Cyclic Sequence Numbers in Label Index Block in the Seq description, and then writing the Index Block over the inactive Index Block location (making this location the new active Index Block if the write succeeds)
Creating a Non-Local Namespace

When creating a new Non-Local Namespace, the software shall follow these steps:

1. Pre-conditions: the labels to be written to each NVDIMM contributing to the namespace have been constructed, each with a unique Position field from 0 to NLabel - 1, and all labels with the same new UUID. All Index Blocks involved have at least 2 label slots free as described in the Label Rules Description and Label Description sections.

2. For each label in the set, the label is written with the EFI_NVDIMM_LABEL_FLAGS_UPDATING flag set, using the flow outlined in Writing New Labels Description to its corresponding NVDIMM / Label Storage Area.

3. For each label in the set, the label is updated with the same contents as the previous step, but with the EFI_NVDIMM_LABEL_FLAGS_UPDATING flag clear, using the flow outlined in Updating an Existing Label Description.

In the case of an unexpected system interruption, the above flows leave either a partial set of labels, all with the new UUID, with the EFI_NVDIMM_LABEL_FLAGS_UPDATING flag set, or a complete set of labels is left where some of them have the EFI_NVDIMM_LABEL_FLAGS_UPDATING flag set. The recovery steps in Recovery Steps for a Non-Local Label Set Description comprehend these two cases so that software can determine whether the set is consistent or needs to be invalidated.

Creating a Local Namespace

Updating labels that are all on the same NVDIMM is atomic with respect to system interruption by nature of the Index Block update rules. Since Local namespaces reside on a single NVDIMM, the EFI_NVDIMM_LABEL_FLAGS_UPDATING flag and multi-pass update described in the previous section are not used. Software creating new Local namespaces shall follow these steps:

1. Pre-conditions: the labels to be written to the NVDIMM Label Storage Area have been constructed, whereby Position, NLabel and SetCookie adhere to the validation rules described earlier, and all labels share the same UUID. The Index Blocks involved have at least NLabel + 1 label slots free, so that after the new labels are written, it will have at least 1 free label slot left.

All labels are written to free slots and made active in one step using steps similar to the flow described above in Writing New Labels Description:

a. Free slots are identified using the current Index Block, the Slot field in each label is updated accordingly

b. All new labels are written into their free slots

c. The new Index Block is constructed so the new label slots are no longer marked free, the sequence number is advanced as shown in Figure Z: Cyclic Sequence Numbers in Label Index Block in the Seq description, and then the new Index Block is written over the inactive Index Block location (making this location the new active Index Block if the write succeeds)

Updating the Name of a Namespace Description

Updating Local Labels

When updating the name on a Local set the sequence outlined in Writing New Labels Description must be followed where the Name is updated before writing the updated Label.
Updating Non Local Labels
To update the Name field associated with a non-Local Namespace, the software must follow these steps:

1. Pre-conditions: the namespace must already exist. Each NVDIMM in the namespace must have at least 1 free slot.
2. For each NVDIMM in the namespace, the label on that NVDIMM is updated with a label with the new Name field and the EFI_NVDIMM_LABEL_FLAGS_UPDATING flag set. The “for each NVDIMM” operation in this step must start with the NVDIMM containing the label whose Position field is zero.
3. For each NVDIMM in the namespace, the label is updated with the same contents as the previous step, but with the EFI_NVDIMM_LABEL_FLAGS_UPDATING flag clear, using the updating an existing label flow described above in Updating an Existing Label Description.

If the above steps are interrupted unexpectedly, the recovery steps in Recovery Steps for a Non-Local Label Set Description handle the case where a Name update is incomplete and finish the update.

13.20 EFI UFS Device Config Protocol

**EFI_UFS_DEVICE_CONFIG_PROTOCOL**

**Summary**
User invokes this protocol to access the UFS device descriptors/flags/attributes and configure UFS device behavior.

**GUID**
```
#define EFI_UFSDEVICE_CONFIG_GUID \ 
{ 0xb81bfab0, 0xeb3, 0x4cf9, \ 
 { 0x84, 0x65, 0x7f, 0xa9, 0x86, 0x36, 0x16, 0x64}}
```

**Protocol Interface Structure**
```
typedef struct __EFI_UFS_DEVICE_CONFIG_PROTOCOL {
  EFI_UFSDEVICE_CONFIG_RW_DESCRIPTOR RwUfsDescriptor;
  EFI_UFSDEVICE_CONFIG_RW_FLAG RwUfsFlag;
  EFI_UFSDEVICE_CONFIG_RW_ATTRIBUTE RwUfsAttribute;
} EFI_UFSDEVICE_CONFIG_PROTOCOL;
```

**Members**
- **RwUfsDescriptor**
  Read or write specified device descriptor of a UFS device.
- **RwUfsFlag**
  Read or write specified flag of a UFS device.
- **RwUfsAttribute**
  Read or write specified attribute of a UFS device.
Description
This protocol aims at defining a standard interface for UEFI drivers and applications to access UFS device descriptors/flags/attributes and configure the UFS device behavior.

**EFI_UFS_DEVICE_CONFIG_PROTOCOL.RwUfsDescriptor()**

Summary
This function is used to read or write specified device descriptor of a UFS device.

Prototype
```c
typedef EFI_STATUS (EFIAPI *EFI_UFS_DEVICE_CONFIG_RW_DESCRIPTOR) (IN EFI_UFSDEVICE_CONFIG_PROTOCOL *This, IN BOOLEAN Read, IN UINT8 DescId, IN UINT8 Index, IN UINT8 Selector, IN OUT UINT8 Descriptor, IN OUT UINT32 *DescSize);
```

Parameters
- **This** The pointer to the EFI_UFSDEVICE_CONFIG_PROTOCOL instance.
- **Read** The boolean variable to show r/w direction.
- **DescId** The ID of device descriptor.
- **Index** The Index of device descriptor.
- **Selector** The Selector of device descriptor.
- **Descriptor** The buffer of device descriptor to be read or written.
- **DescSize** The size of device descriptor buffer. On input, the size, in bytes, of the data buffer specified by Descriptor. On output, the number of bytes that were actually transferred.

Description
The **RwUfsDescriptor** function is used to read/write UFS device descriptors. The consumer of this API is responsible for allocating the data buffer pointed by **Descriptor**.
Status Codes Returned

<table>
<thead>
<tr>
<th>Code</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>EFI_SUCCESS</td>
<td>The device descriptor is read/written successfully.</td>
</tr>
<tr>
<td>EFI_INVALID_PARAMETER</td>
<td>This is NULL or Descriptor is NULL or DescSize is NULL.</td>
</tr>
<tr>
<td>EFI_INVALID_PARAMETER</td>
<td>DescId, Index and Selector are invalid combination to point to a type of UFS device descriptor.</td>
</tr>
<tr>
<td>EFI_DEVICE_ERROR</td>
<td>The device descriptor is not read/written successfully.</td>
</tr>
</tbody>
</table>

`EFI_UFS_DEVICE_CONFIG_PROTOCOL.RwUfsFlag()`

**Summary**
This function is used to read or write specified flag of a UFS device.

**Prototype**

```c
typedef EFI_STATUS
  (EFIAPI *EFI_UFS_DEVICE_CONFIG_RW_FLAG) (  
    IN EFI_UFS_DEVICE_CONFIG_PROTOCOL *This,
    IN BOOLEAN Read,
    IN UINT8 FlagId,
    IN OUT UINT8 *Flag,
  );
```

**Parameters**

- **This**: The pointer to the EFI_UFS_DEVICE_CONFIG_PROTOCOL instance.
- **Read**: The boolean variable to show r/w direction.
- **FlagId**: The ID of flag to be read or written.
- **Flag**: The buffer to set or clear flag.

**Description**

The `RwUfsFlag` function is used to read/write UFS flag descriptors. The consumer of this API is responsible for allocating the buffer pointed by `Flag`. The buffer size is 1 byte as UFS flag descriptor is just a single Boolean value that represents a TRUE or FALSE, ‘0’ or ‘1’, ON or OFF type of value.

**Status Codes Returned**

<table>
<thead>
<tr>
<th>Code</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>EFI_SUCCESS</td>
<td>The flag descriptor is set/clear successfully.</td>
</tr>
<tr>
<td>EFI_INVALID_PARAMETER</td>
<td>This is NULL or Flag is NULL.</td>
</tr>
<tr>
<td>EFI_INVALID_PARAMETER</td>
<td>FlagId is an invalid UFS flag ID.</td>
</tr>
<tr>
<td>EFI_DEVICE_ERROR</td>
<td>The flag is not set/clear successfully.</td>
</tr>
</tbody>
</table>

`EFI_UFS_DEVICE_CONFIG_PROTOCOL.RwUfsAttribute()`

**Summary**
This function is used to read or write specified attribute of a UFS device.
Prototype

```c
typedef
EFI_STATUS
(EFI_API *EFI_UFS_DEVICE_CONFIG_RW_ATTRIBUTE) (
    IN EFI_UFS_DEVICE_CONFIG_PROTOCOL *This,
    IN BOOLEAN Read,
    IN UINT8 AttrId,
    IN UINT8 Index,
    IN UINT8 Selector,
    IN OUT UINT8 *Attribute,
    IN OUT UINT32 *AttrSize,
);
```

Parameters

- **This**: The pointer to the EFI_UFS_DEVICE_CONFIG_PROTOCOL instance.
- **Read**: The boolean variable to show r/w direction.
- **AttrId**: The ID of Attribute.
- **Index**: The Index of Attribute.
- **Selector**: The Selector of Attribute.
- **Attribute**: The buffer of Attribute to be read or written.
- **AttrSize**: The size of Attribute buffer. On input, the size, in bytes, of the data buffer specified by Attribute. On output, the number of bytes that were actually transferred.

Description

The **RwUfsAttribute** function is used to read/write UFS attributes. The consumer of this API is responsible for allocating the data buffer pointed by **Attribute**.

Status Codes Returned

<table>
<thead>
<tr>
<th>Status Code</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>EFI_SUCCESS</td>
<td>The attribute is read/written successfully.</td>
</tr>
<tr>
<td>EFI_INVALID_PARAMETER</td>
<td>This is NULL or Attribute is NULL or AttrSize is NULL.</td>
</tr>
<tr>
<td>EFI_INVALID_PARAMETER</td>
<td>AttrId, Index and Selector are invalid combination to point to a type of UFS attribute.</td>
</tr>
<tr>
<td>EFIDEVICE_ERROR</td>
<td>The attribute is not read/written successfully.</td>
</tr>
</tbody>
</table>
14 - Protocols — PCI Bus Support

14.1 PCI Root Bridge I/O Support

Section 14.1 and Section 14.2 describe the PCI Root Bridge I/O Protocol. This protocol provides an I/O abstraction for a PCI Root Bridge that is produced by a PCI Host Bus Controller. A PCI Host Bus Controller is a hardware component that allows access to a group of PCI devices that share a common pool of PCI I/O and PCI Memory resources. This protocol is used by a PCI Bus Driver to perform PCI Memory, PCI I/O, and PCI Configuration cycles on a PCI Bus. It also provides services to perform different types of bus mastering DMA on a PCI bus. PCI device drivers will not directly use this protocol. Instead, they will use the I/O abstraction produced by the PCI Bus Driver. Only drivers that require direct access to the entire PCI bus should use this protocol. In particular, this chapter defines functions for managing PCI buses, although other bus types may be supported in a similar fashion as extensions to this specification.

All the services described in this chapter that generate PCI transactions follow the ordering rules defined in the PCI Specification. If the processor is performing a combination of PCI transactions and system memory transactions, then there is no guarantee that the system memory transactions will be strongly ordered with respect to the PCI transactions. If strong ordering is required, then processor-specific mechanisms may be required to guarantee strong ordering. Some 64-bit systems may require the use of memory fences to guarantee ordering.

14.1.1 PCI Root Bridge I/O Overview

The interfaces provided in the EFI_PCI_ROOT_BRIDGE_IO_PROTOCOL are for performing basic operations to memory, I/O, and PCI configuration space. The system provides abstracted access to basic system resources to allow a driver to have a programmatic method to access these basic system resources.

The EFI_PCI_ROOT_BRIDGE_IO_PROTOCOL allows for future innovation of the platform. It abstracts device-specific code from the system memory map. This allows system designers to make changes to the system memory map without impacting platform independent code that is consuming basic system resources.

A platform can be viewed as a set of processors and a set of core chipset components that may produce one or more host buses. Figure 14-1 shows a platform with $n$ processors (CPUs in the figure), and a set of core chipset components that produce $m$ host bridges.
Simple systems with one PCI Host Bus Controller will contain a single instance of the
EFI_PCI_ROOT_BRIDGE_IO_PROTOCOL. More complex systems may contain multiple instances of this
protocol. It is important to note that there is no relationship between the number of chip components
in a platform and the number of EFI_PCI_ROOT_BRIDGE_IO_PROTOCOL instances. This protocol
abstracts access to a PCI Root Bridge from a software point of view, and it is attached to a device handle
that represents a PCI Root Bridge. A PCI Root Bridge is a chip component(s) that produces a physical
PCI Bus. It is also the parent to a set of PCI devices that share common PCI I/O, PCI Memory, and PCI
Prefetchable Memory regions. A PCI Host Bus Controller is composed of one or more PCI Root Bridges.

A PCI Host Bridge and PCI Root Bridge are different than a PCI Segment. A PCI Segment is a collection of
up to 256 PCI busses that share the same PCI Configuration Space. Depending on the chipset, a single
EFI_PCI_ROOT_BRIDGE_IO_PROTOCOL may abstract a portion of a PCI Segment, or an entire PCI
Segment. A PCI Host Bridge may produce one or more PCI Root Bridges. When a PCI Host Bridge produces
multiple PCI Root Bridges, it is possible to have more than one PCI Segment.

PCI Root Bridge I/O Protocol instances are either produced by the system firmware or by a UEFI driver.
When a PCI Root Bridge I/O Protocol is produced, it is placed on a device handle along with an EFI Device
Path Protocol instance. Figure 14-2 shows a sample device handle for a PCI Root Bridge Controller that
includes an instance of the EFI_DEVICE_PATH_PROTOCOL and the
EFI_PCI_ROOT_BRIDGE_IO_PROTOCOL. Section 14.2 describes the PCI Root Bridge I/O Protocol in
detail, and Section 14.2.1 describes how to build device paths for PCI Root Bridges. The
EFI_PCI_ROOT_BRIDGE_IO_PROTOCOL does not abstract access to the chipset-specific registers that
are used to manage a PCI Root Bridge. This functionality is hidden within the system firmware or the
driver that produces the handles that represent the PCI Root Bridges.
14.1.1.1 Sample PCI Architectures

The PCI Root Bridge I/O Protocol is designed to provide a software abstraction for a wide variety of PCI architectures including the ones described in this section. This section is not intended to be an exhaustive list of the PCI architectures that the PCI Root Bridge I/O Protocol can support. Instead, it is intended to show the flexibility of this protocol to adapt to current and future platform designs.

Figure 14-3 shows an example of a PCI Host Bus with one PCI Root Bridge. This PCI Root Bridge produces one PCI Local Bus that can contain PCI Devices on the motherboard and/or PCI slots. This would be typical of a desktop system. A higher end desktop system might contain a second PCI Root Bridge for AGP devices. The firmware for this platform would produce one instance of the PCI Root Bridge I/O Protocol.
Figure 14-4 shows an example of a larger server with one PCI Host Bus and four PCI Root Bridges. The PCI devices attached to the PCI Root Bridges are all part of the same coherency domain. This means they share a common PCI I/O Space, a common PCI Memory Space, and a common PCI Prefetchable Memory Space. Each PCI Root Bridge produces one PCI Local Bus that can contain PCI Devices on the motherboard or PCI slots. The firmware for this platform would produce four instances of the PCI Root Bridge I/O Protocol.

Figure 14-5 shows an example of a server with one PCI Host Bus and two PCI Root Bridges. Each of these PCI Root Bridges is a different PCI Segment which allows the system to have up to 512 PCI Buses. A single PCI Segment is limited to 256 PCI Buses. These two segments do not share the same PCI Configuration Space, but they do share the same PCI I/O, PCI Memory, and PCI Prefetchable Memory Space. This is why it can be described by a single PCI Host Bus. The firmware for this platform would produce two instances of the PCI Root Bridge I/O Protocol.
Figure 14-5 Server System with Two PCI Segments

Figure 14-6 shows a server system with two PCI Host Buses and one PCI Root Bridge per PCI Host Bus. This system supports up to 512 PCI Buses, but the PCI I/O, PCI Memory Space, and PCI Prefetchable Memory Space are not shared between the two PCI Root Bridges. The firmware for this platform would produce two instances of the PCI Root Bridge I/O Protocol.
14.2 PCI Root Bridge I/O Protocol

This section provides detailed information on the PCI Root Bridge I/O Protocol and its functions.

**EFI_PCI_ROOT_BRIDGE_IO_PROTOCOL**

**Summary**

Provides the basic Memory, I/O, PCI configuration, and DMA interfaces that are used to abstract accesses to PCI controllers behind a PCI Root Bridge Controller.

**GUID**

```c
#define EFI_PCI_ROOT_BRIDGE_IO_PROTOCOL_GUID
{0x2F707EBB,0x4A1A,0x11d4,
{0x9A,0x38,0x00,0x90,0x27,0x3F,0xC1,0x4D}}
```

**Protocol Interface Structure**

```c
typedef struct _EFI_PCI_ROOT_BRIDGE_IO_PROTOCOL {
  EFI_HANDLE ParentHandle;
  EFI_PCI_ROOT_BRIDGE_IO_PROTOCOL_POLL_IO_MEM PollMem;
  EFI_PCI_ROOT_BRIDGE_IO_PROTOCOL_POLL_IO_MEM PollIo;
  EFI_PCI_ROOT_BRIDGE_IO_PROTOCOL_ACCESS Mem;
  EFI_PCI_ROOT_BRIDGE_IO_PROTOCOL_ACCESS Io;
  EFI_PCI_ROOT_BRIDGE_IO_PROTOCOL_ACCESS Pci;
  EFI_PCI_ROOT_BRIDGE_IO_PROTOCOL_COPY_MEM CopyMem;
  EFI_PCI_ROOT_BRIDGE_IO_PROTOCOL_MAP Map;
  EFI_PCI_ROOT_BRIDGE_IO_PROTOCOL_UNMAP Unmap;
  EFI_PCI_ROOT_BRIDGE_IO_PROTOCOL_ALLOCATE_BUFFER AllocateBuffer;
  EFI_PCI_ROOT_BRIDGE_IO_PROTOCOL_FREE_BUFFER FreeBuffer;
  EFI_PCI_ROOT_BRIDGE_IO_PROTOCOL_FLUSH Flush;
  EFI_PCI_ROOT_BRIDGE_IO_PROTOCOL_GET_ATTRIBUTES GetAttributes;
  EFI_PCI_ROOT_BRIDGE_IO_PROTOCOL_SET_ATTRIBUTES SetAttributes;
  EFI_PCI_ROOT_BRIDGE_IO_PROTOCOL_CONFIGURATION Configuration;
  UINT32 SegmentNumber;
} EFI_PCI_ROOT_BRIDGE_IO_PROTOCOL;
```

**Parameters**

- **ParentHandle**
  The **EFI_HANDLE** of the PCI Host Bridge of which this PCI Root Bridge is a member.

- **PollMem**
  Polls an address in memory mapped I/O space until an exit condition is met, or a timeout occurs. See the **PollMem()** function description.

- **PollIo**
  Polls an address in I/O space until an exit condition is met, or a timeout occurs. See the **PollIo()** function description.

- **Mem.Read**
  Allows reads from memory mapped I/O space. See the **Mem.Read()** function description.
Mem.Write
Returns writes to memory mapped I/O space. See the Mem.Write() function description.

Io.Read
Returns reads from I/O space. See the Io.Read() function description.

Io.Write
Returns writes to I/O space. See the Io.Write() function description.

Pci.Read
Returns reads from PCI configuration space. See the Pci.Read() function description.

Pci.Write
Returns writes to PCI configuration space. See the Pci.Write() function description.

CopyMem
Returns one region of PCI root bridge memory space to be copied to another region of PCI root bridge memory space. See the CopyMem() function description.

Map
Provides the PCI controller-specific addresses needed to access system memory for DMA. See the Map() function description.

Unmap
Releases any resources allocated by Map(). See the Unmap() function description.

AllocateBuffer
Allocates pages that are suitable for a common buffer mapping. See the AllocateBuffer() function description.

FreeBuffer
Free pages that were allocated with AllocateBuffer(). See the FreeBuffer() function description.

Flush
Flushes all PCI posted write transactions to system memory. See the Flush() function description.

GetAttributes
Gets the attributes that a PCI root bridge supports setting with SetAttributes(), and the attributes that a PCI root bridge is currently using. See the GetAttributes() function description.

SetAttributes
Sets attributes for a resource range on a PCI root bridge. See the SetAttributes() function description.

Configuration
Gets the current resource settings for this PCI root bridge. See the Configuration() function description.

SegmentNumber
The segment number that this PCI root bridge resides.

Related Definitions

```c
typedef enum {
    EfiPciWidthUint8,
    EfiPciWidthUint16,
    EfiPciWidthUint32,
    EfiPciWidthUint64,
    EfiPciWidthFifoUint8,
    EfiPciWidthFifoUint16,
    EfiPciWidthFifoUint32,
    EfiPciWidthFifoUint64,
} EFI_PCI_ROOT_BRIDGE_IO_PROTOCOL_WIDTH;
```
EfiPciWidthFillUint8,
EfiPciWidthFillUint16,
EfiPciWidthFillUint32,
EfiPciWidthFillUint64,
EfiPciWidthMaximum
} EFI_PCI_ROOT_BRIDGE_IO_PROTOCOL_WIDTH;

//*******************************************************
// EFI_PCI_ROOT_BRIDGE_IO_PROTOCOL_POLL_IO_MEM
//*******************************************************
typedef
EFI_STATUS
(EIFIAPI *EFI_PCI_ROOT_BRIDGE_IO_PROTOCOL_POLL_IO_MEM) ( 
  IN struct EFI_PCI_ROOT_BRIDGE_IO_PROTOCOL *This,
  IN EFI_PCI_ROOT_BRIDGE_IO_PROTOCOL_WIDTH Width,
  IN UINT64 Address,
  IN UINT64 Mask,
  IN UINT64 Value,
  IN UINT64 Delay,
  OUT UINT64 *Result
);

//*******************************************************
// EFI_PCI_ROOT_BRIDGE_IO_PROTOCOL_IO_MEM
//*******************************************************
typedef
EFI_STATUS
(EIFIAPI *EFI_PCI_ROOT_BRIDGE_IO_PROTOCOL_IO_MEM) ( 
  IN   EFI_PCI_ROOT_BRIDGE_IO_PROTOCOL *This,
  IN   EFI_PCI_ROOT_BRIDGE_IO_PROTOCOL_WIDTH Width,
  IN   UINT64 Address,
  IN   UINTN Count,
  IN OUT VOID *Buffer
);

//*******************************************************
// EFI_PCI_ROOT_BRIDGE_IO_PROTOCOL_ACCESS
//*******************************************************
typedef struct {
  EFI_PCI_ROOT_BRIDGE_IO_PROTOCOL_IO_MEM Read;
  EFI_PCI_ROOT_BRIDGE_IO_PROTOCOL_IO_MEM Write;
} EFI_PCI_ROOT_BRIDGE_IO_PROTOCOL_ACCESS;

//******************************************************************************
//EFI PCI Root Bridge I/O Protocol Attribute bits
//******************************************************************************
#define EFI_PCI_ATTRIBUTE_ISA_MOTHERBOARD_IO  0x0001


#define EFI_PCI_ATTRIBUTE_ISA_IO              0x0002
#define EFI_PCI_ATTRIBUTE_VGA_PALETTE_IO        0x0004
#define EFI_PCI_ATTRIBUTE_VGA_MEMORY            0x0008
#define EFI_PCI_ATTRIBUTE_VGA_IO                0x0010
#define EFI_PCI_ATTRIBUTE_IDE_PRIMARY_IO        0x0020
#define EFI_PCI_ATTRIBUTE_IDE_SECONDARY_IO      0x0040
#define EFI_PCI_ATTRIBUTE_MEMORY_WRITE_COMBINE  0x0080
#define EFI_PCI_ATTRIBUTE_MEMORY_CACHED         0x0800
#define EFI_PCI_ATTRIBUTE_MEMORY_DISABLE        0x1000
#define EFI_PCI_ATTRIBUTE_DUAL_ADDRESS_CYCLE    0x8000
#define EFI_PCI_ATTRIBUTE_ISA_IO_16             0x10000
#define EFI_PCI_ATTRIBUTE_VGA_PALETTE_IO_16     0x20000
#define EFI_PCI_ATTRIBUTE_VGA_IO_16             0x40000

EFI_PCI_ATTRIBUTE_ISA_IO_16
If this bit is set, then the PCI I/O cycles between 0x100 and 0x3FF are forwarded onto a PCI root bridge using a 16-bit address decoder on address bits 0..15. Address bits 16..31 must be zero. This bit is used to forward I/O cycles for legacy ISA devices onto a PCI root bridge. This bit may not be combined with EFI_PCI_ATTRIBUTE_ISA_IO.

EFI_PCI_ATTRIBUTE_VGA_PALETTE_IO_16
If this bit is set, then the PCI I/O write cycles for 0x3C6, 0x3C8, and 0x3C9 are forwarded onto a PCI root bridge using a 16-bit address decoder on address bits 0..15. Address bits 16..31 must be zero. This bit is used to forward I/O write cycles to the VGA palette registers onto a PCI root bridge. This bit may not be combined with EFI_PCI_ATTRIBUTE_VGA_IO or EFI_PCI_ATTRIBUTE_VGA_PALETTE_IO.

EFI_PCI_ATTRIBUTE_VGA_IO_16
If this bit is set, then the PCI I/O cycles in the ranges 0x3B0–0x3BB and 0x3C0–0x3DF are forwarded onto a PCI root bridge using a 16-bit address decoder on address bits 0..15. Address bits 16..31 must be zero. This bit is used to forward I/O cycles for a VGA controller onto a PCI root bridge. This bit may not be combined with EFI_PCI_ATTRIBUTE_VGA_IO or EFI_PCI_ATTRIBUTE_VGA_PALETTE_IO. Because EFI_PCI_ATTRIBUTE_VGA_IO_16 also includes the I/O range described by EFI_PCI_ATTRIBUTE_VGA_PALETTE_IO_16, the EFI_PCI_ATTRIBUTE_VGA_PALETTE_IO_16 bit is ignored if EFI_PCI_ATTRIBUTE_VGA_IO_16 is set.

EFI_PCI_ATTRIBUTE_ISA_MOTHERBOARD_IO
If this bit is set, then the PCI I/O cycles between 0x00000000 and 0x000000FF are forwarded onto a PCI root bridge. This bit is used to forward I/O cycles for ISA motherboard devices onto a PCI root bridge.

EFI_PCI_ATTRIBUTE_ISA_IO
If this bit is set, then the PCI I/O cycles between 0x100 and 0x3FF are forwarded onto a PCI root bridge using a 10-bit address decoder
on address bits 0..9. Address bits 10..15 are not decoded, and address bits 16..31 must be zero. This bit is used to forward I/O cycles for legacy ISA devices onto a PCI root bridge.

**EFI_PCI_ATTRIBUTE_VGA_PALETTE_IO**
If this bit is set, then the PCI I/O write cycles for 0x3C6, 0x3C8, and 0x3C9 are forwarded onto a PCI root bridge using a 10 bit address decoder on address bits 0..9. Address bits 10..15 are not decoded, and address bits 16..31 must be zero. This bit is used to forward I/O write cycles to the VGA palette registers onto a PCI root bridge.

**EFI_PCI_ATTRIBUTE_VGA_MEMORY**
If this bit is set, then the PCI memory cycles between 0xA0000 and 0xBFFFF are forwarded onto a PCI root bridge. This bit is used to forward memory cycles for a VGA frame buffer onto a PCI root bridge.

**EFI_PCI_ATTRIBUTE_VGA_IO**
If this bit is set, then the PCI I/O cycles in the ranges 0x3B0-0x3BB and 0x3C0-0x3DF are forwarded onto a PCI root bridge using a 10-bit address decoder on address bits 0..9. Address bits 10..15 are not decoded, and the address bits 16..31 must be zero. This bit is used to forward I/O cycles for a VGA controller onto a PCI root bridge. Since *EFI_PCI_ATTRIBUTE_ENABLE_VGA_IO* also includes the I/O range described by *EFI_PCI_ATTRIBUTE_ENABLE_VGA_PALETTE_IO*, the *EFI_PCI_ATTRIBUTE_ENABLE_VGA_PALETTE_IO* bit is ignored if *EFI_PCI_ATTRIBUTE_ENABLE_VGA_IO* is set.

**EFI_PCI_ATTRIBUTE_IDE_PRIMARY_IO**
If this bit is set, then the PCI I/O cycles in the ranges 0x1F0-0x1F7 and 0x3F6-0x3F7 are forwarded onto a PCI root bridge using a 16-bit address decoder on address bits 0..15. Address bits 16..31 must be zero. This bit is used to forward I/O cycles for a Primary IDE controller onto a PCI root bridge.

**EFI_PCI_ATTRIBUTE_IDE_SECONDARY_IO**
If this bit is set, then the PCI I/O cycles in the ranges 0x170-0x177 and 0x376-0x377 are forwarded onto a PCI root bridge using a 16-bit address decoder on address bits 0..15. Address bits 16..31 must be zero. This bit is used to forward I/O cycles for a Secondary IDE controller onto a PCI root bridge.

**EFI_PCI_ATTRIBUTE_MEMORY_WRITE_COMBINE**
If this bit is set, then this platform supports changing the attributes of a PCI memory range so that the memory range is accessed in a write combining mode. By default, PCI memory ranges are not accessed in a write combining mode.

**EFI_PCI_ATTRIBUTE_MEMORY_CACHED**
If this bit is set, then this platform supports changing the attributes of a PCI memory range so that the memory range is accessed in a cached mode. By default, PCI memory ranges are accessed noncached.

**EFI_PCI_ATTRIBUTE_MEMORY_DISABLE**
If this bit is set, then this platform supports changing the attributes
of a PCI memory range so that the memory range is disabled, and can no longer be accessed. By default, all PCI memory ranges are enabled.

EFI_PCI_ATTRIBUTE_DUAL_ADDRESS_CYCLE
This bit may only be used in the Attributes parameter to AllocateBuffer(). If this bit is set, then the PCI controller that is requesting a buffer through AllocateBuffer() is capable of producing PCI Dual Address Cycles, so it is able to access a 64-bit address space. If this bit is not set, then the PCI controller that is requesting a buffer through AllocateBuffer() is not capable of producing PCI Dual Address Cycles, so it is only able to access a 32-bit address space.

typedef enum {
    EfiPciOperationBusMasterRead,
    EfiPciOperationBusMasterWrite,
    EfiPciOperationBusMasterCommonBuffer,
    EfiPciOperationBusMasterRead64,
    EfiPciOperationBusMasterWrite64,
    EfiPciOperationBusMasterCommonBuffer64,
    EfiPciOperationMaximum
} EFI_PCI_ROOT_BRIDGE_IO_PROTOCOL_OPERATION;

EfiPciOperationBusMasterRead
A read operation from system memory by a bus master that is not capable of producing PCI dual address cycles.

EfiPciOperationBusMasterWrite
A write operation to system memory by a bus master that is not capable of producing PCI dual address cycles.

EfiPciOperationBusMasterCommonBuffer
Provides both read and write access to system memory by both the processor and a bus master that is not capable of producing PCI dual address cycles. The buffer is coherent from both the processor’s and the bus master’s point of view.

EfiPciOperationBusMasterRead64
A read operation from system memory by a bus master that is capable of producing PCI dual address cycles.

EfiPciOperationBusMasterWrite64
A write operation to system memory by a bus master that is capable of producing PCI dual address cycles.
EfiPciOperationBusMasterCommonBuffer64

Provides both read and write access to system memory by both the processor and a bus master that is capable of producing PCI dual address cycles. The buffer is coherent from both the processor’s and the bus master’s point of view.

Description

The EFI_PCI_ROOT_BRIDGE_IO_PROTOCOL provides the basic Memory, I/O, PCI configuration, and DMA interfaces that are used to abstract accesses to PCI controllers. There is one EFI_PCI_ROOT_BRIDGE_IO_PROTOCOL instance for each PCI root bridge in a system. Embedded systems, desktops, and workstations will typically only have one PCI root bridge. High-end servers may have multiple PCI root bridges. A device driver that wishes to manage a PCI bus in a system will have to retrieve the EFI_PCI_ROOT_BRIDGE_IO_PROTOCOL instance that is associated with the PCI bus to be managed. A device handle for a PCI Root Bridge will minimally contain an EFI_DEVICE_PATH_PROTOCOL instance and an EFI_PCI_ROOT_BRIDGE_IO_PROTOCOL instance. The PCI bus driver can look at the EFI_DEVICE_PATH_PROTOCOL instances to determine which EFI_PCI_ROOT_BRIDGE_IO_PROTOCOL instance to use.

Bus mastering PCI controllers can use the DMA services for DMA operations. There are three basic types of bus mastering DMA that is supported by this protocol. These are DMA reads by a bus master, DMA writes by a bus master, and common buffer DMA. The DMA read and write operations may need to be broken into smaller chunks. The caller of Map() must pay attention to the number of bytes that were mapped, and if required, loop until the entire buffer has been transferred. The following is a list of the different bus mastering DMA operations that are supported, and the sequence of EFI_PCI_ROOT_BRIDGE_IO_PROTOCOL APIs that are used for each DMA operation type. See “Related Definitions” above for the definition of the different DMA operation types.

DMA Bus Master Read Operation

- Call Map() for EfiPciOperationBusMasterRead or EfiPciOperationBusMasterRead64.
- Program the DMA Bus Master with the DeviceAddress returned by Map().
- Start the DMA Bus Master.
- Wait for DMA Bus Master to complete the read operation.
- Call Unmap().

DMA Bus Master Write Operation

- Call Map() for EfiPciOperationBusMasterWrite or EfiPciOperationBusMasterRead64.
- Program the DMA Bus Master with the DeviceAddress returned by Map().
- Start the DMA Bus Master.
- Wait for DMA Bus Master to complete the write operation.
- Perform a PCI controller specific read transaction to flush all PCI write buffers (See PCI Specification Section 3.2.5.2).
- Call Flush().
- Call Unmap().

DMA Bus Master Common Buffer Operation
• Call **AllocateBuffer()** to allocate a common buffer.
• Call **Map()** for **EfiPciOperationBusMasterCommonBuffer** or **EfiPciOperationBusMasterCommonBuffer64**.
• Program the DMA Bus Master with the **DeviceAddress** returned by **Map()**.
• The common buffer can now be accessed equally by the processor and the DMA bus master.
• Call **Unmap()**.
• Call **FreeBuffer()**.

**EFI_PCI_ROOT_BRIDGE_IO_PROTOCOL.PollMem()**

**Summary**
Reads from the memory space of a PCI Root Bridge. Returns when either the polling exit criteria is satisfied or after a defined duration.

**Prototype**
```
typedef EFI_STATUS
    (EFIAPI *EFI_PCI_ROOT_BRIDGE_IO_PROTOCOL_POLL_IO_MEM) (  
    IN EFI_PCI_ROOT_BRIDGE_IO_PROTOCOL *This,  
    IN EFI_PCI_ROOT_BRIDGE_IO_PROTOCOL_WIDTH Width,  
    IN UINT64 Address,  
    IN UINT64 Mask,  
    IN UINT64 Value,  
    IN UINT64 Delay,  
    OUT UINT64 *Result  
    );
```

**Parameters**
- **This**
  A pointer to the **EFI_PCI_ROOT_BRIDGE_IO_PROTOCOL**. Type **EFI_PCI_ROOT_BRIDGE_IO_PROTOCOL** is defined in [Section 14.2](#).
- **Width**
  Signifies the width of the memory operations. Type **EFI_PCI_ROOT_BRIDGE_IO_PROTOCOL_WIDTH** is defined in [Section 14.2](#).
- **Address**
  The base address of the memory operations. The caller is responsible for aligning **Address** if required.
- **Mask**
  Mask used for the polling criteria. Bytes above **Width** in **Mask** are ignored. The bits in the bytes below **Width** which are zero in **Mask** are ignored when polling the memory address.
- **Value**
  The comparison value used for the polling exit criteria.
- **Delay**
  The number of 100 ns units to poll. Note that timer available may be of poorer granularity.
- **Result**
  Pointer to the last value read from the memory location.
Description
This function provides a standard way to poll a PCI memory location. A PCI memory read operation is performed at the PCI memory address specified by Address for the width specified by Width. The result of this PCI memory read operation is stored in Result. This PCI memory read operation is repeated until either a timeout of Delay 100 ns units has expired, or (Result & Mask) is equal to Value.

This function will always perform at least one PCI memory read access no matter how small Delay may be. If Delay is zero, then Result will be returned with a status of EFI_SUCCESS even if Result does not match the exit criteria. If Delay expires, then EFI_TIMEOUT is returned.

If Width is not EfiPciWidthUint8, EfiPciWidthUint16, EfiPciWidthUint32, or EfiPciWidthUint64, then EFI_INVALID_PARAMETER is returned.

The memory operations are carried out exactly as requested. The caller is responsible for satisfying any alignment and memory width restrictions that a PCI Root Bridge on a platform might require. For example on some platforms, width requests of EfiPciWidthUint64 are not supported.

All the PCI transactions generated by this function are guaranteed to be completed before this function returns. However, if the memory mapped I/O region being accessed by this function has the EFI_PCI_ATTRIBUTE_MEMORY_CACHED attribute set, then the transactions will follow the ordering rules defined by the processor architecture.

Status Codes Returned

<table>
<thead>
<tr>
<th>Code</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>EFI_SUCCESS</td>
<td>The last data returned from the access matched the poll exit criteria.</td>
</tr>
<tr>
<td>EFI_INVALID_PARAMETER</td>
<td>Width is invalid.</td>
</tr>
<tr>
<td>EFI_INVALID_PARAMETER</td>
<td>Result is NULL.</td>
</tr>
<tr>
<td>EFI_TIMEOUT</td>
<td>Delay expired before a match occurred.</td>
</tr>
<tr>
<td>EFI_OUT_OF_RESOURCES</td>
<td>The request could not be completed due to a lack of resources.</td>
</tr>
</tbody>
</table>

EFI_PCI_ROOT_BRIDGE_IO_PROTOCOL.PollIo()

Summary
Reads from the I/O space of a PCI Root Bridge. Returns when either the polling exit criteria is satisfied or after a defined duration.
Prototype

typedef

EFI_STATUS

(EIFIAPI *EFI_PCI_ROOT_BRIDGE_IO_PROTOCOL_POLL_IO_MEM) (  
    IN EFI_PCI_ROOT_BRIDGE_IO_PROTOCOL *This,
    IN EFI_PCI_ROOT_BRIDGE_IO_PROTOCOL_WIDTH Width,
    IN UINT64 Address,
    IN UINT64 Mask,
    IN UINT64 Value,
    IN UINT64 Delay,
    OUT UINT64 *Result
);

Parameters

This A pointer to the EFI_PCI_ROOT_BRIDGE_IO_PROTOCOL. Type EFI_PCI_ROOT_BRIDGE_IO_PROTOCOL is defined in Section 14.2.

Width Signifies the width of the I/O operations. Type EFI_PCI_ROOT_BRIDGE_IO_PROTOCOL_WIDTH is defined in Section 14.2.

Address The base address of the I/O operations. The caller is responsible for aligning Address if required.

Mask Mask used for the polling criteria. Bytes above Width in Mask are ignored. The bits in the bytes below Width which are zero in Mask are ignored when polling the I/O address.

Value The comparison value used for the polling exit criteria.

Delay The number of 100 ns units to poll. Note that timer available may be of poorer granularity.

Result Pointer to the last value read from the memory location.

Description

This function provides a standard way to poll a PCI I/O location. A PCI I/O read operation is performed at the PCI I/O address specified by Address for the width specified by Width. The result of this PCI I/O read operation is stored in Result. This PCI I/O read operation is repeated until either a timeout of Delay 100 ns units has expired, or (Result & Mask) is equal to Value.

This function will always perform at least one I/O access no matter how small Delay may be. If Delay is zero, then Result will be returned with a status of EFI_SUCCESS even if Result does not match the exit criteria. If Delay expires, then EFI_TIMEOUT is returned.

If Width is not EfiPciWidthUint8, EfiPciWidthUint16, EfiPciWidthUint32, or EfiPciWidthUint64, then EFI_INVALID_PARAMETER is returned.

The I/O operations are carried out exactly as requested. The caller is responsible satisfying any alignment and I/O width restrictions that the PCI Root Bridge on a platform might require. For example on some platforms, width requests of EfiPciWidthUint64 do not work.

All the PCI transactions generated by this function are guaranteed to be completed before this function returns.
Status Codes Returned

<table>
<thead>
<tr>
<th>Code</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>EFI_SUCCESS</td>
<td>The last data returned from the access matched the poll exit criteria.</td>
</tr>
<tr>
<td>EFI_INVALID_PARAMETER</td>
<td>Width is invalid.</td>
</tr>
<tr>
<td>EFI_INVALID_PARAMETER</td>
<td>Result is NULL.</td>
</tr>
<tr>
<td>EFI_TIMEOUT</td>
<td>Delay expired before a match occurred.</td>
</tr>
<tr>
<td>EFI_OUT_OF_RESOURCES</td>
<td>The request could not be completed due to a lack of resources.</td>
</tr>
</tbody>
</table>

**EFI_PCI_ROOT_BRIDGE_IO_PROTOCOL.Mem.Read()**

**EFI_PCI_ROOT_BRIDGE_IO_PROTOCOL.Mem.Write()**

**Summary**

Enables a PCI driver to access PCI controller registers in the PCI root bridge memory space.

**Prototype**

```c
typedef
EFI_STATUS
(EFIAPI *EFI_PCI_ROOT_BRIDGE_IO_PROTOCOL_IO_MEM) (  
  IN   EFI_PCI_ROOT_BRIDGE_IO_PROTOCOL   *This,  
  IN   EFI_PCI_ROOT_BRIDGE_IO_PROTOCOL_WIDTH Width,  
  IN   UINT64                           Address,  
  IN   UINTN                            Count,  
  IN OUT VOID                           *Buffer
);```

**Parameters**

- **This**
  - A pointer to the **EFI_PCI_ROOT_BRIDGE_IO_PROTOCOL** Type
  - **EFI_PCI_ROOT_BRIDGE_IO_PROTOCOL** is defined in [Section 14.2](#).

- **Width**
  - Signifies the width of the memory operation. Type
  - **EFI_PCI_ROOT_BRIDGE_IO_PROTOCOL_WIDTH** is defined in [Section 14.2](#).

- **Address**
  - The base address of the memory operation. The caller is responsible for aligning the **Address** if required.

- **Count**
  - The number of memory operations to perform. Bytes moved is **Width** size * **Count**, starting at **Address**.

- **Buffer**
  - For read operations, the destination buffer to store the results. For write operations, the source buffer to write data from.

**Description**

The **Mem.Read()**, and **Mem.Write()** functions enable a driver to access PCI controller registers in the PCI root bridge memory space.

The memory operations are carried out exactly as requested. The caller is responsible for satisfying any alignment and memory width restrictions that a PCI Root Bridge on a platform might require. For example on some platforms, width requests of **EfiPciWidthUint64** do not work.
If Width is `EfiPciWidthUint8`, `EfiPciWidthUint16`, `EfiPciWidthUint32`, or `EfiPciWidthUint64`, then both Address and Buffer are incremented for each of the Count operations performed.

If Width is `EfiPciWidthFifoUint8`, `EfiPciWidthFifoUint16`, `EfiPciWidthFifoUint32`, or `EfiPciWidthFifoUint64`, then only Buffer is incremented for each of the Count operations performed. The read or write operation is performed Count times on the same Address.

If Width is `EfiPciWidthFillUint8`, `EfiPciWidthFillUint16`, `EfiPciWidthFillUint32`, or `EfiPciWidthFillUint64`, then only Address is incremented for each of the Count operations performed. The read or write operation is performed Count times from the first element of Buffer.

All the PCI read transactions generated by this function are guaranteed to be completed before this function returns. All the PCI write transactions generated by this function will follow the write ordering and completion rules defined in the PCI Specification. However, if the memory-mapped I/O region being accessed by this function has the `EFI_PCI_ATTRIBUTE_MEMORY_CACHED` attribute set, then the transactions will follow the ordering rules defined by the processor architecture.

<table>
<thead>
<tr>
<th>Status Codes Returned</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>EFI_SUCCESS</strong></td>
</tr>
<tr>
<td>The data was read from or written to the PCI root bridge.</td>
</tr>
<tr>
<td><strong>EFI_INVALID_PARAMETER</strong></td>
</tr>
<tr>
<td>Width is invalid for this PCI root bridge.</td>
</tr>
<tr>
<td><strong>EFI_INVALID_PARAMETER</strong></td>
</tr>
<tr>
<td>Buffer is NULL.</td>
</tr>
<tr>
<td><strong>EFI_OUT_OF_RESOURCES</strong></td>
</tr>
<tr>
<td>The request could not be completed due to a lack of resources.</td>
</tr>
</tbody>
</table>

**EFI_PCI_ROOT_BRIDGE_IO_PROTOCOL.Io.Read()**  
**EFI_PCI_ROOT_BRIDGE_IO_PROTOCOL.Io.Write()**

**Summary**
Enables a PCI driver to access PCI controller registers in the PCI root bridge I/O space.

**Prototype**
```c
typedef EFI_STATUS (EFIAPI *EFI_PCI_ROOT_BRIDGE_IO_PROTOCOL_IO_MEM) (  
  IN   EFI_PCI_ROOT_BRIDGE_IO_PROTOCOL *This,  
  IN   EFI_PCI_ROOT_BRIDGE_IO_PROTOCOL_WIDTH Width,  
  IN   UINT64 Address,  
  IN   UINTN Count,  
  IN OUT VOID *Buffer
);
```

**Parameters**
- **This**  
  A pointer to the EFI_PCI_ROOT_BRIDGE_IO_PROTOCOL Type.
- **Width**  
  Signifies the width of the memory operations. Type `EFI_PCI_ROOT_BRIDGE_IO_PROTOCOL_WIDTH` is defined in Section 14.2.
Address
The base address of the I/O operation. The caller is responsible for aligning the Address if required.

Count
The number of I/O operations to perform. Bytes moved is Width size * Count, starting at Address.

Buffer
For read operations, the destination buffer to store the results. For write operations, the source buffer to write data from.

Description
The Io.Read(), and Io.Write() functions enable a driver to access PCI controller registers in the PCI root bridge I/O space.

The I/O operations are carried out exactly as requested. The caller is responsible for satisfying any alignment and I/O width restrictions that a PCI root bridge on a platform might require. For example on some platforms, width requests of EfiPciWidthUint64 do not work.

If Width is EfiPciWidthUint8, EfiPciWidthUint16, EfiPciWidthUint32, or EfiPciWidthUint64, then both Address and Buffer are incremented for each of the Count operations performed.

If Width is EfiPciWidthFifoUint8, EfiPciWidthFifoUint16, EfiPciWidthFifoUint32, or EfiPciWidthFifoUint64, then only Buffer is incremented for each of the Count operations performed. The read or write operation is performed Count times on the same Address.

If Width is EfiPciWidthFillUint8, EfiPciWidthFillUint16, EfiPciWidthFillUint32, or EfiPciWidthFillUint64, then only Address is incremented for each of the Count operations performed. The read or write operation is performed Count times from the first element of Buffer.

All the PCI transactions generated by this function are guaranteed to be completed before this function returns.

Status Codes Returned

<table>
<thead>
<tr>
<th>Code</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>EFI_SUCCESS</td>
<td>The data was read from or written to the PCI root bridge.</td>
</tr>
<tr>
<td>EFI_INVALID_PARAMETER</td>
<td>Width is invalid for this PCI root bridge.</td>
</tr>
<tr>
<td>EFI_INVALID_PARAMETER</td>
<td>Buffer is NULL.</td>
</tr>
<tr>
<td>EFI_OUT_OF_RESOURCES</td>
<td>The request could not be completed due to a lack of resources.</td>
</tr>
</tbody>
</table>

EFI_PCI_ROOT_BRIDGE_IO_PROTOCOL.Pci.Read()
EFI_PCI_ROOT_BRIDGE_IO_PROTOCOL.Pci.Write()

Summary
Enables a PCI driver to access PCI controller registers in a PCI root bridge’s configuration space.
Prototype

```c
typedef EFI_STATUS
  (EFIAPIC *EFI_PCI_ROOT_BRIDGE_IO_PROTOCOL_IO_MEM)
  (IN EFI_PCI_ROOT_BRIDGE_IO_PROTOCOL *
This,
IN EFI_PCI_ROOT_BRIDGE_IO_PROTOCOL_WIDTH Width,
IN UINT64 Address,
IN UINTN Count,
IN OUT VOID *Buffer);
```

Parameters

- **This**: A pointer to the EFI_PCI_ROOT_BRIDGE_IO_PROTOCOL. Type EFI_PCI_ROOT_BRIDGE_IO_PROTOCOL is defined in Section 14.2.
- **Width**: Signifies the width of the memory operations. Type EFI_PCI_ROOT_BRIDGE_IO_PROTOCOL_WIDTH is defined in Section 14.2.
- **Address**: The address within the PCI configuration space for the PCI controller. See Table 14-1 for the format of Address.
- **Count**: The number of PCI configuration operations to perform. Bytes moved is Width size * Count, starting at Address.
- **Buffer**: For read operations, the destination buffer to store the results. For write operations, the source buffer to write data from.

Description

The **Pci.Read()** and **Pci.Write()** functions enable a driver to access PCI configuration registers for a PCI controller.

The PCI Configuration operations are carried out exactly as requested. The caller is responsible for any alignment and PCI configuration width issues that a PCI Root Bridge on a platform might require. For example on some platforms, width requests of EfiPciWidthUint64 do not work.

If **Width** is EfiPciWidthUint8, EfiPciWidthUint16, EfiPciWidthUint32, or EfiPciWidthUint64, then both **Address** and **Buffer** are incremented for each of the **Count** operations performed.

If **Width** is EfiPciWidthFifoUint8, EfiPciWidthFifoUint16, EfiPciWidthFifoUint32, or EfiPciWidthFifoUint64, then only **Buffer** is incremented for each of the **Count** operations performed. The read or write operation is performed **Count** times on the same **Address**.

If **Width** is EfiPciWidthFillUint8, EfiPciWidthFillUint16, EfiPciWidthFillUint32, or EfiPciWidthFillUint64, then only **Address** is incremented for each of the **Count** operations performed. The read or write operation is performed **Count** times from the first element of **Buffer**.

All the PCI transactions generated by this function are guaranteed to be completed before this function returns.
Table 14-1 PCI Configuration Address

<table>
<thead>
<tr>
<th>Mnemonic</th>
<th>Byte Offset</th>
<th>Byte Length</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Register</td>
<td>0</td>
<td>1</td>
<td>The register number on the PCI Function.</td>
</tr>
<tr>
<td>Function</td>
<td>1</td>
<td>1</td>
<td>The PCI Function number on the PCI Device.</td>
</tr>
<tr>
<td>Device</td>
<td>2</td>
<td>1</td>
<td>The PCI Device number on the PCI Bus.</td>
</tr>
<tr>
<td>Bus</td>
<td>3</td>
<td>1</td>
<td>The PCI Bus number.</td>
</tr>
<tr>
<td>ExtendedRegister</td>
<td>4</td>
<td>4</td>
<td>The register number on the PCI Function. If this field is zero, then the Register field is used for the register number. If this field is nonzero, then the Register field is ignored, and the ExtendedRegister field is used for the register number.</td>
</tr>
</tbody>
</table>

Status Codes Returned

<table>
<thead>
<tr>
<th>Code</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>EFI_SUCCESS</td>
<td>The data was read from or written to the PCI root bridge.</td>
</tr>
<tr>
<td>EFI_INVALID_PARAMETER Width</td>
<td>Width is invalid for this PCI root bridge.</td>
</tr>
<tr>
<td>EFI_INVALID_PARAMETER Buffer</td>
<td>Buffer is NULL.</td>
</tr>
<tr>
<td>EFI_OUT_OF_RESOURCES</td>
<td>The request could not be completed due to a lack of resources.</td>
</tr>
</tbody>
</table>

EFI_PCI_ROOT_BRIDGE_IO_PROTOCOL.CopyMem()

Summary

Enables a PCI driver to copy one region of PCI root bridge memory space to another region of PCI root bridge memory space.

Prototype

```c
typedef
    EFI_STATUS
(EFIAPI *EFI_PCI_ROOT_BRIDGE_IO_PROTOCOL_COPY_MEM) (  
    IN EFI_PCI_ROOT_BRIDGE_IO_PROTOCOL *This,
    IN EFI_PCI_ROOT_BRIDGE_IO_PROTOCOL_WIDTH Width,
    IN UINT64 DestAddress,
    IN UINT64 SrcAddress,
    IN UINTN Count
    );
```

Parameters

- **This**
  A pointer to the EFI_PCI_ROOT_BRIDGE_IO_PROTOCOL instance. Type **EFI_PCI_ROOT_BRIDGE_IO_PROTOCOL** is defined in **Section 14.2**.

- **Width**
  Signifies the width of the memory operations. Type **EFI_PCI_ROOT_BRIDGE_IO_PROTOCOL_WIDTH** is defined in **Section 14.2**.

- **DestAddress**
  The destination address of the memory operation. The caller is responsible for aligning the **DestAddress** if required.
**SrcAddress**
The source address of the memory operation. The caller is responsible for aligning the *SrcAddress* if required.

**Count**
The number of memory operations to perform. Bytes moved is *Width* size * Count, starting at *DestAddress* and *SrcAddress*.

**Description**
The CopyMem() function enables a PCI driver to copy one region of PCI root bridge memory space to another region of PCI root bridge memory space. This is especially useful for video scroll operation on a memory mapped video buffer.

The memory operations are carried out exactly as requested. The caller is responsible for satisfying any alignment and memory width restrictions that a PCI root bridge on a platform might require. For example on some platforms, width requests of EfiPciWidthUint64 do not work.

If *Width* is EfiPciIoWidthUint8, EfiPciIoWidthUint16, EfiPciIoWidthUint32, or EfiPciIoWidthUint64, then *Count* read/write transactions are performed to move the contents of the *SrcAddress* buffer to the *DestAddress* buffer. The implementation must be reentrant, and it must handle overlapping *SrcAddress* and *DestAddress* buffers. This means that the implementation of CopyMem() must choose the correct direction of the copy operation based on the type of overlap that exists between the *SrcAddress* and *DestAddress* buffers. If either the *SrcAddress* buffer or the *DestAddress* buffer crosses the top of the processor’s address space, then the result of the copy operation is unpredictable.

The contents of the *DestAddress* buffer on exit from this service must match the contents of the *SrcAddress* buffer on entry to this service. Due to potential overlaps, the contents of the *SrcAddress* buffer may be modified by this service. The following rules can be used to guarantee the correct behavior:

- If *DestAddress* > *SrcAddress* and *DestAddress* < ( *SrcAddress* + *Width* size * Count), then the data should be copied from the *SrcAddress* buffer to the *DestAddress* buffer starting from the end of buffers and working toward the beginning of the buffers.
- Otherwise, the data should be copied from the *SrcAddress* buffer to the *DestAddress* buffer starting from the beginning of the buffers and working toward the end of the buffers.

All the PCI transactions generated by this function are guaranteed to be completed before this function returns. All the PCI write transactions generated by this function will follow the write ordering and completion rules defined in the PCI Specification. However, if the memory-mapped I/O region being accessed by this function has the EFI_PCI_ATTRIBUTE_MEMORY_CACHED attribute set, then the transactions will follow the ordering rules defined by the processor architecture.
Status Codes Returned

<table>
<thead>
<tr>
<th>Code</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>EFI_SUCCESS</td>
<td>The data was copied from one memory region to another memory region.</td>
</tr>
<tr>
<td>EFI_INVALID_PARAMETER</td>
<td>Width is invalid for this PCI root bridge.</td>
</tr>
<tr>
<td>EFI_OUT_OF_RESOURCES</td>
<td>The request could not be completed due to a lack of resources.</td>
</tr>
</tbody>
</table>

**EFI_PCI_ROOT_BRIDGE_IO_PROTOCOL.Map()**

**Summary**

Provides the PCI controller–specific addresses required to access system memory from a DMA bus master.

**Prototype**

```c
typedef EFI_STATUS (EFIAPI *EFI_PCI_ROOT_BRIDGE_IO_PROTOCOL_MAP)(
  IN   EFI_PCI_ROOT_BRIDGE_IO_PROTOCOL *This,
  IN   EFI_PCI_ROOT_BRIDGE_IO_PROTOCOL_OPERATION Operation,
  IN   VOID *HostAddress,
  IN OUT UINTN *NumberOfBytes,
  OUT  EFI_PHYSICAL_ADDRESS *DeviceAddress,
  OUT  VOID **Mapping
);
```

**Parameters**

- **This**
  A pointer to the EFI_PCI_ROOT_BRIDGE_IO_PROTOCOL. Type EFI_PCI_ROOT_BRIDGE_IO_PROTOCOL is defined in Section 14.2.

- **Operation**
  Indicates if the bus master is going to read or write to system memory. Type EFI_PCI_ROOT_BRIDGE_IO_PROTOCOL_OPERATION is defined in Section 14.2.

- **HostAddress**
  The system memory address to map to the PCI controller.

- **NumberOfBytes**
  On input the number of bytes to map. On output the number of bytes that were mapped.

- **DeviceAddress**
  The resulting map address for the bus master PCI controller to use to access the system memory’s HostAddress. Type EFI_PHYSICAL_ADDRESS is defined in EFI_BOOT_SERVICES.AllocatePages(). This address cannot be used by the processor to access the contents of the buffer specified by HostAddress.

- **Mapping**
  The value to pass to Unmap() when the bus master DMA operation is complete.

**Description**

The Map() function provides the PCI controller specific addresses needed to access system memory. This function is used to map system memory for PCI bus master DMA accesses.
All PCI bus master accesses must be performed through their mapped addresses and such mappings must be freed with `Unmap()` when complete. If the bus master access is a single read or single write data transfer, then `EfiPciOperationBusMasterRead`, `EfiPciOperationBusMasterRead64`, `EfiPciOperationBusMasterWrite`, or `EfiPciOperationBusMasterWrite64` is used and the range is unmapped to complete the operation. If performing an `EfiPciOperationBusMasterRead` or `EfiPciOperationBusMasterRead64` operation, all the data must be present in system memory before `Map()` is performed. Similarly, if performing an `EfiPciOperation-BusMasterWrite` or `EfiPciOperationBusMasterWrite64` the data cannot be properly accessed in system memory until `Unmap()` is performed.

Bus master operations that require both read and write access or require multiple host device interactions within the same mapped region must use `EfiPciOperation-BusMasterCommonBuffer` or `EfiPciOperationBusMasterCommonBuffer64`. However, only memory allocated via the `AllocateBuffer()` interface can be mapped for this type of operation.

In all mapping requests the resulting `NumberOfBytes` actually mapped may be less than the requested amount. In this case, the DMA operation will have to be broken up into smaller chunks. The `Map()` function will map as much of the DMA operation as it can at one time. The caller may have to loop on `Map()` and `Unmap()` in order to complete a large DMA transfer.

### Status Codes Returned

<table>
<thead>
<tr>
<th>Status Code</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>EFI_SUCCESS</code></td>
<td>The range was mapped for the returned <code>NumberOfBytes</code>.</td>
</tr>
<tr>
<td><code>EFI_INVALID_PARAMETER</code></td>
<td><code>Operation</code> is invalid.</td>
</tr>
<tr>
<td><code>EFI_INVALID_PARAMETER</code></td>
<td><code>HostAddress</code> is NULL.</td>
</tr>
<tr>
<td><code>EFI_INVALID_PARAMETER</code></td>
<td><code>NumberOfBytes</code> is NULL.</td>
</tr>
<tr>
<td><code>EFI_INVALID_PARAMETER</code></td>
<td><code>DeviceAddress</code> is NULL.</td>
</tr>
<tr>
<td><code>EFI_INVALID_PARAMETER</code></td>
<td><code>Mapping</code> is NULL.</td>
</tr>
<tr>
<td><code>EFI_UNSUPPORTED</code></td>
<td><code>HostAddress</code> cannot be mapped as a common buffer.</td>
</tr>
<tr>
<td><code>EFI_DEVICE_ERROR</code></td>
<td>The system hardware could not map the requested address.</td>
</tr>
<tr>
<td><code>EFI_OUT_OF_RESOURCES</code></td>
<td>The request could not be completed due to a lack of resources.</td>
</tr>
</tbody>
</table>

### EFI_PCI_ROOT_BRIDGE_IO_PROTOCOL.Unmap()

**Summary**

Completes the `Map()` operation and releases any corresponding resources.
Prototype

```c
typedef
EFI_STATUS
(EFI_API *EFI_PCI_ROOT_BRIDGE_IO_PROTOCOL_UNMAP) (
    IN EFI_PCI_ROOT_BRIDGE_IO_PROTOCOL *This,
    IN VOID *Mapping
);
```

Parameters

- **This**: A pointer to the `EFI_PCI_ROOT_BRIDGE_IO_PROTOCOL` Type. `EFI_PCI_ROOT_BRIDGE_IO_PROTOCOL` is defined in Section 14.2.
- **Mapping**: The mapping value returned from `Map()`.

Description

The `Unmap()` function completes the `Map()` operation and releases any corresponding resources. If the operation was an `EfiPciOperationBusMasterWrite` or `EfiPciOperationBusMasterWrite64`, the data is committed to the target system memory. Any resources used for the mapping are freed.

Status Codes Returned

<table>
<thead>
<tr>
<th>Status Code</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>EFI_SUCCESS</td>
<td>The range was unmapped.</td>
</tr>
</tbody>
</table>
| EFI_INVALID_PARAMETER        | `Mapping` is not a value that was returned by `Map()`.
| EFI_DEVICE_ERROR             | The data was not committed to the target system memory. |

**`EFI_PCI_ROOT_BRIDGE_IO_PROTOCOL.AllocateBuffer()`**

Summary

Allocates pages that are suitable for an `EfiPciOperationBusMasterCommonBuffer` or `EfiPciOperationBusMasterCommonBuffer64` mapping.

Prototype

```c
typedef
EFI_STATUS
(EFI_API *EFI_PCI_ROOT_BRIDGE_IO_PROTOCOL_ALLOCATE_BUFFER) (
    IN EFI_PCI_ROOT_BRIDGE_IO_PROTOCOL *This,
    IN EFI_ALLOCATE_TYPE Type,
    IN EFI_MEMORY_TYPE MemoryType,
    IN UINTN Pages,
    OUT VOID **HostAddress,
    IN UINT64 Attributes
);
```

Parameters

- **This**: A pointer to the `EFI_PCI_ROOT_BRIDGE_IO_PROTOCOL` Type. `EFI_PCI_ROOT_BRIDGE_IO_PROTOCOL` is defined in Section 14.2.1.
- **Type**: This parameter is not used and must be ignored.
MemoryType

The type of memory to allocate, EfiBootServicesData or EfiRuntimeServicesData. Type EFI_MEMORY_TYPE is defined in EFI_BOOT_SERVICES.AllocatePages().

Pages

The number of pages to allocate.

HostAddress

A pointer to store the base system memory address of the allocated range.

Attributes

The requested bit mask of attributes for the allocated range. Only the attributes EFI_PCI_ATTRIBUTE_MEMORY_WRITE_COMBINE, EFI_PCI_ATTRIBUTE_MEMORY_CACHED, and EFI_PCI_ATTRIBUTE_DUAL_ADDRESS_CYCLE may be used with this function. If any other bits are set, then EFI_UNSUPPORTED is returned. This function may choose to ignore this bit mask. The EFI_PCI_ATTRIBUTE_MEMORY_WRITE_COMBINE, and EFI_PCI_ATTRIBUTE_MEMORY_CACHED attributes provide a hint to the implementation that may improve the performance of the calling driver. The implementation may choose any default for the memory attributes including write combining, cached, both, or neither as long as the allocated buffer can be seen equally by both the processor and the PCI bus master.

Description

The AllocateBuffer() function allocates pages that are suitable for an EfiPciOperationBusMasterCommonBuffer or EfiPciOperationBusMasterCommonBuffer64 mapping. This means that the buffer allocated by this function must support simultaneous access by both the processor and a PCI Bus Master. The device address that the PCI Bus Master uses to access the buffer can be retrieved with a call to Map().

If the EFI_PCI_ATTRIBUTE_DUAL_ADDRESS_CYCLE bit of Attributes is set, then when the buffer allocated by this function is mapped with a call to Map(), the device address that is returned by Map() must be within the 64-bit device address space of the PCI Bus Master.

If the EFI_PCI_ATTRIBUTE_DUAL_ADDRESS_CYCLE bit of Attributes is clear, then when the buffer allocated by this function is mapped with a call to Map(), the device address that is returned by Map() must be within the 32-bit device address space of the PCI Bus Master.

If the memory allocation specified by MemoryType and Pages cannot be satisfied, then EFI_OUT_OF_RESOURCES is returned.
Status Codes Returned

<table>
<thead>
<tr>
<th>Status Code</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>EFI_SUCCESS</td>
<td>The requested memory pages were allocated.</td>
</tr>
<tr>
<td>EFI_INVALID_PARAMETER</td>
<td>MemoryType is invalid.</td>
</tr>
<tr>
<td>EFI_INVALID_PARAMETER</td>
<td>HostAddress is NULL.</td>
</tr>
<tr>
<td>EFI_UNSUPPORTED</td>
<td>Attributes is unsupported. The only legal attribute bits are</td>
</tr>
<tr>
<td></td>
<td>EFI_PCI_ATTRIBUTE_MEMORY_WRITE_COMBINE, EFI_PCI_ATTRIBUTE_MEMORY_CACHED, and</td>
</tr>
<tr>
<td></td>
<td>EFI_PCI_ATTRIBUTE_DUAL_ADDRESS_CYCLE.</td>
</tr>
<tr>
<td>EFI_OUT_OF_RESOURCES</td>
<td>The memory pages could not be allocated.</td>
</tr>
</tbody>
</table>

**EFI_PCI_ROOT_BRIDGE_IO_PROTOCOL.FreeBuffer()**

**Summary**
Frees memory that was allocated with AllocateBuffer().

**Prototype**

```c
typedef EFI_STATUS (EFIAPI *EFI_PCI_ROOT_BRIDGE_IO_PROTOCOL_FREE_BUFFER) (
    IN EFI_PCI_ROOT_BRIDGE_IO_PROTOCOL *This,
    IN UINTN Pages,
    IN VOID *HostAddress
);
```

**Parameters**

- **This**
  A pointer to the EFI_PCI_ROOT_BRIDGE_IO_PROTOCOL. Type `EFI_PCI_ROOT_BRIDGE_IO_PROTOCOL` is defined in Section 14.2.

- **Pages**
  The number of pages to free.

- **HostAddress**
  The base system memory address of the allocated range.

**Description**

The FreeBuffer() function frees memory that was allocated with AllocateBuffer().

**Status Codes Returned**

<table>
<thead>
<tr>
<th>Status Code</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>EFI_SUCCESS</td>
<td>The requested memory pages were freed.</td>
</tr>
<tr>
<td>EFI_INVALID_PARAMETER</td>
<td>The memory range specified by HostAddress and Pages was not allocated with AllocateBuffer().</td>
</tr>
</tbody>
</table>

**EFI_PCI_ROOT_BRIDGE_IO_PROTOCOL.Flush()**

**Summary**
Flushes all PCI posted write transactions from a PCI host bridge to system memory.
Prototype

```c
typedef
EFI_STATUS
(EFIAPI *EFI_PCI_ROOT_BRIDGE_IO_PROTOCOL_FLUSH) (IN EFI_PCI_ROOT_BRIDGE_IO_PROTOCOL *This);
```

Parameters

- `This`: A pointer to the `EFI_PCI_ROOT_BRIDGE_IO_PROTOCOL` type.

Description

The `Flush()` function flushes any PCI posted write transactions from a PCI host bridge to system memory. Posted write transactions are generated by PCI bus masters when they perform write transactions to target addresses in system memory.

This function does not flush posted write transactions from any PCI bridges. A PCI controller specific action must be taken to guarantee that the posted write transactions have been flushed from the PCI controller and from all the PCI bridges into the PCI host bridge. This is typically done with a PCI read transaction from the PCI controller prior to calling `Flush()`.

If the PCI controller specific action required to flush the PCI posted write transactions has been performed, and this function returns `EFI_SUCCESS`, then the PCI bus master’s view and the processor’s view of system memory are guaranteed to be coherent. If the PCI posted write transactions cannot be flushed from the PCI host bridge, then the PCI bus master and processor are not guaranteed to have a coherent view of system memory, and `EFI_DEVICE_ERROR` is returned.

Status Codes Returned

<table>
<thead>
<tr>
<th>Status Code</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>EFI_SUCCESS</td>
<td>The PCI posted write transactions were flushed from the PCI host bridge to system memory.</td>
</tr>
<tr>
<td>EFI_DEVICE_ERROR</td>
<td>The PCI posted write transactions were not flushed from the PCI host bridge due to a hardware error.</td>
</tr>
</tbody>
</table>

`EFI_PCI_ROOT_BRIDGE_IO_PROTOCOL.GetAttributes()`

Summary

Gets the attributes that a PCI root bridge supports setting with `SetAttributes()`, and the attributes that a PCI root bridge is currently using.
Prototype

```c
typedef EFI_STATUS
    (EFIAPI *EFI_PCI_ROOT_BRIDGE_IO_PROTOCOL_GET_ATTRIBUTES) (
        IN EFI_PCI_ROOT_BRIDGE_IO_PROTOCOL *This,
        OUT UINT64 *Supports OPTIONAL,
        OUT UINT64 *Attributes OPTIONAL
    );
```

Parameters

- **This**: A pointer to the EFI_PCI_ROOT_BRIDGE_IO_PROTOCOL. Type EFI_PCI_ROOT_BRIDGE_IO_PROTOCOL is defined in Section 14.2.
- **Supports**: A pointer to the mask of attributes that this PCI root bridge supports setting with SetAttributes(). The available attributes are listed in Section 14.2. This is an optional parameter that may be NULL.
- **Attributes**: A pointer to the mask of attributes that this PCI root bridge is currently using. The available attributes are listed in Section 14.2. This is an optional parameter that may be NULL.

Description

The GetAttributes() function returns the mask of attributes that this PCI root bridge supports and the mask of attributes that the PCI root bridge is currently using. If Supports is not NULL, then Supports is set to the mask of attributes that the PCI root bridge supports. If Attributes is not NULL, then Attributes is set to the mask of attributes that the PCI root bridge is currently using. If both Supports and Attributes are NULL, then EFI_INVALID_PARAMETER is returned. Otherwise, EFI_SUCCESS is returned.

If a bit is set in Supports, then the PCI root bridge supports this attribute type, and a call can be made to SetAttributes() using that attribute type. If a bit is set in Attributes, then the PCI root bridge is currently using that attribute type. Since a PCI host bus may be composed of more than one PCI root bridge, different Attributes values may be returned by different PCI root bridges.

Status Codes Returned

<table>
<thead>
<tr>
<th>Code</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>EFI_SUCCESS</td>
<td>If Supports is not NULL, then the attributes that the PCI root bridge supports are returned in Supports. If Attributes is not NULL, then the attributes that the PCI root bridge is currently using is returned in Attributes.</td>
</tr>
<tr>
<td>EFI_INVALID_PARAMETER</td>
<td>Both Supports and Attributes are NULL.</td>
</tr>
</tbody>
</table>

EFI_PCI_ROOT_BRIDGE_IO_PROTOCOL.SetAttributes()

Summary

Sets attributes for a resource range on a PCI root bridge.
Prototype

```
typedef
EFI_STATUS
(EIFIAPI *EFI_PCI_ROOT_BRIDGE_IO_PROTOCOL_SET_ATTRIBUTES) (
    IN EFI_PCI_ROOT_BRIDGE_IO_PROTOCOL *This,
    IN UINT64 Attributes,
    IN OUT UINT64 *ResourceBase OPTIONAL,
    IN OUT UINT64 *ResourceLength OPTIONAL
);
```

Parameters

- **This**
  A pointer to the EFI_PCI_ROOT_BRIDGE_IO_PROTOCOL Type.
  EFI_PCI_ROOT_BRIDGE_IO_PROTOCOL is defined in Section 14.2.

- **Attributes**
  The mask of attributes to set. If the attribute bit
  MEMORY_WRITE_COMBINE, MEMORY_CACHED, or
  MEMORY_DISABLE is set, then the resource range is specified by
  ResourceBase and ResourceLength. If MEMORY_WRITE_COMBINE,
  MEMORY_CACHED, and MEMORY_DISABLE are not set, then
  ResourceBase and ResourceLength are ignored, and may be NULL.
  The available attributes are listed in Section 14.2.

- **ResourceBase**
  A pointer to the base address of the resource range to be modified
  by the attributes specified by Attributes. On return, *ResourceBase
  will be set the actual base address of the resource range. Not all
  resources can be set to a byte boundary, so the actual base address
  may differ from the one passed in by the caller. This parameter is
  only used if the MEMORY_WRITE_COMBINE bit, the
  MEMORY_CACHED bit, or the MEMORY_DISABLE bit of Attributes is
  set. Otherwise, it is ignored, and may be NULL.

- **ResourceLength**
  A pointer to the length of the resource range to be modified by the
  attributes specified by Attributes. On return, *ResourceLength
  will be set the actual length of the resource range. Not all resources
  can be set to a byte boundary, so the actual length may differ from
  the one passed in by the caller. This parameter is only used if the
  MEMORY_WRITE_COMBINE bit, the MEMORY_CACHED bit, or the
  MEMORY_DISABLE bit of Attributes is set. Otherwise, it is ignored,
  and may be NULL.

Description

The **SetAttributes()** function sets the attributes specified in Attributes for the PCI root bridge on
the resource range specified by **ResourceBase** and **ResourceLength**. Since the granularity of setting
these attributes may vary from resource type to resource type, and from platform to platform, the actual
resource range and the one passed in by the caller may differ. As a result, this function may set the
attributes specified by Attributes on a larger resource range than the caller requested. The actual
range is returned in **ResourceBase** and **ResourceLength**. The caller is responsible for verifying that
the actual range for which the attributes were set is acceptable.

If the attributes are set on the PCI root bridge, then the actual resource range is returned in **ResourceBase** and **ResourceLength**, and **EFI_SUCCESS** is returned.
If the attributes specified by `Attributes` are not supported by the PCI root bridge, then `EFI_UNSUPPORTED` is returned. The set of supported attributes for a PCI root bridge can be found by calling `GetAttributes()`.

If either `ResourceBase` or `ResourceLength` are `NULL`, and a resource range is required for the attributes specified in `Attributes`, then `EFI_INVALID_PARAMETER` is returned.

If more than one resource range is required for the set of attributes specified by `Attributes`, then `EFI_INVALID_PARAMETER` is returned.

If there are not enough resources available to set the attributes, then `EFI_OUT_OF_RESOURCES` is returned.

### Status Codes Returned

<table>
<thead>
<tr>
<th>Status Code</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>EFI_SUCCESS</td>
<td>The set of attributes specified by <code>Attributes</code> for the resource range specified by <code>ResourceBase</code> and <code>ResourceLength</code> were set on the PCI root bridge, and the actual resource range is returned in <code>ResourceBase</code> and <code>ResourceLength</code>.</td>
</tr>
<tr>
<td>EFI_UNSUPPORTED</td>
<td>A bit is set in <code>Attributes</code> that is not supported by the PCI Root Bridge. The supported attribute bits are reported by <code>GetAttributes()</code>.</td>
</tr>
<tr>
<td>EFI_INVALID_PARAMETER</td>
<td>More than one attribute bit is set in <code>Attributes</code> that requires a resource range.</td>
</tr>
<tr>
<td>EFI_INVALID_PARAMETER</td>
<td>A resource range is required, and <code>ResourceBase</code> is <code>NULL</code>.</td>
</tr>
<tr>
<td>EFI_INVALID_PARAMETER</td>
<td>A resource range is required, and <code>ResourceLength</code> is <code>NULL</code>.</td>
</tr>
<tr>
<td>EFI_OUT_OF_RESOURCES</td>
<td>There are not enough resources to set the attributes on the resource range specified by <code>BaseAddress</code> and <code>Length</code>.</td>
</tr>
</tbody>
</table>

### EFI_PCI_ROOT_BRIDGE_IO_PROTOCOL.Configuration()

#### Summary

Retrieves the current resource settings of this PCI root bridge in the form of a set of ACPI resource descriptors.

#### Prototype

```c
typedef EFI_STATUS (EFIAPICallingConvention) EFI_PCI_ROOT_BRIDGE_IO_PROTOCOL_CONFIGURATION (IN EFI_PCI_ROOT_BRIDGE_IO_PROTOCOL *This,
                                         OUT VOID **Resources );
```

#### Parameters

- **This**  
  A pointer to the `EFI_PCI_ROOT_BRIDGE_IO_PROTOCOL` Type. `EFI_PCI_ROOT_BRIDGE_IO_PROTOCOL` is defined in [Section 14.2](#).

- **Resources**  
  A pointer to the resource descriptors that describe the current configuration of this PCI root bridge. The storage for the resource descriptors is allocated by this function. The caller must treat the return buffer as read-only data, and the buffer must not be freed by...
the caller. See “Related Definitions” for the resource descriptors that may be used.

Related Definitions

There are only two resource descriptor types from the ACPI Specification that may be used to describe the current resources allocated to a PCI root bridge. These are the QWORD Address Space Descriptor, and the End Tag. The QWORD Address Space Descriptor can describe memory, I/O, and bus number ranges for dynamic or fixed resources. The configuration of a PCI root bridge is described with one or more QWORD Address Space Descriptors followed by an End Tag. Table 14-3 and Table 14-2 contain these two descriptor types.

Please see the ACPI Specification for details on the field values. The definition of the Address Space Granularity field in the QWORD Address Space Descriptor differs from the ACPI Specification, and the definition in Table 14-2 is the one that must be used.

### Table 14-2 QWORD Address Space Descriptor

<table>
<thead>
<tr>
<th>Byte Offset</th>
<th>Byte Length</th>
<th>Data</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x00</td>
<td>0x01</td>
<td>0x8A</td>
<td>QWORD Address Space Descriptor</td>
</tr>
<tr>
<td>0x01</td>
<td>0x02</td>
<td>0x2B</td>
<td>Length of this descriptor in bytes not including the first two fields</td>
</tr>
<tr>
<td>0x03</td>
<td>0x01</td>
<td></td>
<td>Resource Type</td>
</tr>
<tr>
<td></td>
<td></td>
<td>0 – Memory Range</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>1 – I/O Range</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>2 – Bus Number Range</td>
<td></td>
</tr>
<tr>
<td>0x04</td>
<td>0x01</td>
<td></td>
<td>General Flags</td>
</tr>
<tr>
<td>0x05</td>
<td>0x01</td>
<td></td>
<td>Type Specific Flags</td>
</tr>
<tr>
<td>0x06</td>
<td>0x08</td>
<td></td>
<td>Address Space Granularity. Used to differentiate between a 32-bit memory request and a 64-bit memory request. For a 32-bit memory request, this field should be set to 32. For a 64-bit memory request, this field should be set to 64.</td>
</tr>
<tr>
<td>0xE</td>
<td>0x08</td>
<td></td>
<td>Address Range Minimum</td>
</tr>
<tr>
<td>0x16</td>
<td>0x08</td>
<td></td>
<td>Address Range Maximum</td>
</tr>
<tr>
<td>0x1E</td>
<td>0x08</td>
<td></td>
<td>Address Translation Offset. Offset to apply to the Starting address to convert it to a PCI address. This value is zero unless the HostAddress and DeviceAddress for the root bridge are different.</td>
</tr>
<tr>
<td>0x26</td>
<td>0x08</td>
<td></td>
<td>Address Length</td>
</tr>
</tbody>
</table>

### Table 14-3 End Tag

<table>
<thead>
<tr>
<th>Byte Offset</th>
<th>Byte Length</th>
<th>Data</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x00</td>
<td>0x01</td>
<td>0x79</td>
<td>End Tag</td>
</tr>
<tr>
<td>0x01</td>
<td>0x01</td>
<td>0x00</td>
<td>Checksum. If 0, then checksum is assumed to be valid.</td>
</tr>
</tbody>
</table>
Description

The `Configuration()` function retrieves a set of resource descriptors that contains the current configuration of this PCI root bridge. If the current configuration can be retrieved, then it is returned in `Resources` and `EFI_SUCCESS` is returned. See “Related Definitions” below for the resource descriptor types that are supported by this function. If the current configuration cannot be retrieved, then `EFI_UNSUPPORTED` is returned.

### Status Codes Returned

<table>
<thead>
<tr>
<th>Status Code</th>
<th>Message</th>
</tr>
</thead>
<tbody>
<tr>
<td>EFI_SUCCESS</td>
<td>The current configuration of this PCI root bridge was returned in <code>Resources</code>.</td>
</tr>
<tr>
<td>EFI_UNSUPPORTED</td>
<td>The current configuration of this PCI root bridge could not be retrieved.</td>
</tr>
</tbody>
</table>

#### 14.2.1 PCI Root Bridge Device Paths

An `EFI_PCI_ROOT_BRIDGE_IO_PROTOCOL` must be installed on a handle for its services to be available to drivers. In addition to the `EFI_PCI_ROOT_BRIDGE_IO_PROTOCOL`, an `EFI_DEVICE_PATH_PROTOCOL` must also be installed on the same handle.

Typically, an ACPI Device Path Node is used to describe a PCI Root Bridge. Depending on the bus hierarchy in the system, additional device path nodes may precede this ACPI Device Path Node. A desktop system will typically contain only one PCI Root Bridge, so there would be one handle with an `EFI_PCI_ROOT_BRIDGE_IO_PROTOCOL` and an `EFI_DEVICE_PATH_PROTOCOL`. A server system may contain multiple PCI Root Bridges, so it would contain a handle for each PCI Root Bridge present, and on each of those handles would be an `EFI_PCI_ROOT_BRIDGE_IO_PROTOCOL` and an `EFI_DEVICE_PATH_PROTOCOL`. In all cases, the contents of the ACPI Device Path Nodes for PCI Root Bridges must match the information present in the ACPI tables for that system.

*Table 14-4* shows an example device path for a PCI Root Bridge in a desktop system. Today, a desktop system typically contains only one PCI Root Bridge. This device path consists of an ACPI Device Path Node, and a Device Path End Structure. The `_HID` and `_UID` must match the ACPI table description of the PCI Root Bridge. For a system with only one PCI Root Bridge, the `_UID` value is usually 0x0000. The shorthand notation for this device path is `ACPI(PNP0A03,0)`.

**Table 14-4 PCI Root Bridge Device Path for a Desktop System**

<table>
<thead>
<tr>
<th>Byte Offset</th>
<th>Byte Length</th>
<th>Data</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x00</td>
<td>0x01</td>
<td>0x02</td>
<td>Generic Device Path Header – Type ACPI Device Path</td>
</tr>
<tr>
<td>0x01</td>
<td>0x01</td>
<td>0x01</td>
<td>Sub type – ACPI Device Path</td>
</tr>
<tr>
<td>0x02</td>
<td>0x02</td>
<td>0x0C</td>
<td>Length – 0x0C bytes</td>
</tr>
<tr>
<td>0x04</td>
<td>0x04</td>
<td>0x41D0, 0x0A03</td>
<td>_HID PNP0A03 – 0x41D0 represents the compressed string ‘PNP’ and is encoded in the low order bytes. The compression method is described in the ACPI Specification.</td>
</tr>
<tr>
<td>0x08</td>
<td>0x04</td>
<td>0x0000</td>
<td>_UID</td>
</tr>
<tr>
<td>0x0C</td>
<td>0x01</td>
<td>0xFF</td>
<td>Generic Device Path Header – Type End of Hardware Device Path</td>
</tr>
<tr>
<td>0x0D</td>
<td>0x01</td>
<td>0xFF</td>
<td>Sub type – End of Entire Device Path</td>
</tr>
<tr>
<td>0x0E</td>
<td>0x02</td>
<td>0x04</td>
<td>Length – 0x04 bytes</td>
</tr>
</tbody>
</table>
Table 14-5 through Table 14-8 show example device paths for the PCI Root Bridges in a server system with four PCI Root Bridges. Each of these device paths consists of an ACPI Device Path Node, and a Device Path End Structure. The _HID and _UID must match the ACPI table description of the PCI Root Bridges. The only difference between each of these device paths is the _UID field. The shorthand notation for these four device paths is `ACPI(PNP0A03, 0), ACPI(PNP0A03, 1), ACPI(PNP0A03, 2),` and `ACPI(PNP0A03, 3).`
Table 14-8 PCI Root Bridge Device Path for Bridge #3 in a Server System

<table>
<thead>
<tr>
<th>Byte Offset</th>
<th>Byte Length</th>
<th>Data</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x00</td>
<td>0x01</td>
<td>0x02</td>
<td>Generic Device Path Header – Type ACPI Device Path</td>
</tr>
<tr>
<td>0x01</td>
<td>0x01</td>
<td>0x01</td>
<td>Sub type – ACPI Device Path</td>
</tr>
<tr>
<td>0x02</td>
<td>0x02</td>
<td>0x0C</td>
<td>Length – 0x0C bytes</td>
</tr>
<tr>
<td>0x04</td>
<td>0x04</td>
<td>0x41D0, 0x0A03</td>
<td>_HID PNP0A03 – 0x41D0 represents the compressed string ‘PNP’ and is encoded in the low order bytes. The compression method is described in the ACPI Specification.</td>
</tr>
<tr>
<td>0x08</td>
<td>0x04</td>
<td>0x0002</td>
<td>_UID</td>
</tr>
<tr>
<td>0x0C</td>
<td>0x01</td>
<td>0xFF</td>
<td>Generic Device Path Header – Type End of Hardware Device Path</td>
</tr>
<tr>
<td>0x0D</td>
<td>0x01</td>
<td>0xFF</td>
<td>Sub type – End of Entire Device Path</td>
</tr>
<tr>
<td>0x0E</td>
<td>0x02</td>
<td>0x04</td>
<td>Length – 0x04 bytes</td>
</tr>
</tbody>
</table>

Table 14-9 shows an example device path for a PCI Root Bridge using an Expanded ACPI Device Path. This device path consists of an Expanded ACPI Device Path Node, and a Device Path End Structure. The _UID and _CID fields must match the ACPI table description of the PCI Root Bridge. For a system with only one PCI Root Bridge, the _UID value is usually 0x0000. The shorthand notation for this device path is ACPI(12345678,0,PNP0A03).

Table 14-9 PCI Root Bridge Device Path Using Expanded ACPI Device Path

<table>
<thead>
<tr>
<th>Byte Offset</th>
<th>Byte Length</th>
<th>Data</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x00</td>
<td>0x01</td>
<td>0x02</td>
<td>Generic Device Path Header – Type ACPI Device Path</td>
</tr>
<tr>
<td>0x01</td>
<td>0x01</td>
<td>0x02</td>
<td>Sub type – Expanded ACPI Device Path</td>
</tr>
<tr>
<td>0x02</td>
<td>0x02</td>
<td>0x10</td>
<td>Length – 0x10 bytes</td>
</tr>
<tr>
<td>0x04</td>
<td>0x04</td>
<td>0x1234, 0x5678</td>
<td>_HID-device specific</td>
</tr>
<tr>
<td>0x08</td>
<td>0x04</td>
<td>0x0000</td>
<td>_UID</td>
</tr>
<tr>
<td>0x0C</td>
<td>0x04</td>
<td>0x41D0, 0x0A03</td>
<td>_CID PNP0A03 – 0x41D0 represents the compressed string ‘PNP’ and is encoded in the low order bytes. The compression method is described in the ACPI Specification.</td>
</tr>
<tr>
<td>0x10</td>
<td>0x01</td>
<td>0xFF</td>
<td>Generic Device Path Header – Type End of Hardware Device Path</td>
</tr>
<tr>
<td>0x11</td>
<td>0x01</td>
<td>0xFF</td>
<td>Sub type – End of Entire Device Path</td>
</tr>
<tr>
<td>0x12</td>
<td>0x02</td>
<td>0x04</td>
<td>Length – 0x04 bytes</td>
</tr>
</tbody>
</table>
14.3 PCI Driver Model

Section 14.3 and Section 14.4 describe the PCI Driver Model. This includes the behavior of PCI Bus Drivers, the behavior of a PCI Device Drivers, and a detailed description of the PCI I/O Protocol. The PCI Bus Driver manages PCI buses present in a system, and PCI Device Drivers manage PCI controllers present on PCI buses. The PCI Device Drivers produce an I/O abstraction that can be used to boot an EFI compliant operating system.

This document provides enough material to implement a PCI Bus Driver, and the tools required to design and implement a PCI Device Drivers. It does not provide any information on specific PCI devices.

The material contained in this section is designed to extend this specification and the UEFI Driver Model in a way that supports PCI device drivers and PCI bus drivers. These extensions are provided in the form of PCI-specific protocols. This section provides the information required to implement a PCI Bus Driver in system firmware. The section also contains the information required by driver writers to design and implement PCI Device Drivers that a platform may need to boot a UEFI-compliant OS.

The PCI Driver Model described here is intended to be a foundation on which a PCI Bus Driver and a wide variety of PCI Device Drivers can be created.

14.3.1 PCI Driver Initialization

There are very few differences between a PCI Bus Driver and PCI Device Driver in the entry point of the driver. The file for a driver image must be loaded from some type of media. This could include ROM, FLASH, hard drives, floppy drives, CD-ROM, or even a network connection. Once a driver image has been found, it can be loaded into system memory with the Boot Service EFI_BOOT_SERVICES.LoadImage(). LoadImage() loads a PE/COFF formatted image into system memory. A handle is created for the driver, and a Loaded Image Protocol instance is placed on that handle. A handle that contains a Loaded Image Protocol instance is called an Image Handle. At this point, the driver has not been started. It is just sitting in memory waiting to be started. Figure 14-7 shows the state of an image handle for a driver after LoadImage() has been called.

![Image Handle](image-handle.png)

After a driver has been loaded with the Boot Service EFI_BOOT_SERVICES.LoadImage(), it must be started with the Boot Service EFI_BOOT_SERVICES.StartImage(). This is true of all types of applications and drivers that can be loaded and started on an UEFI compliant system. The entry point for a driver that follows the UEFI Driver Model must follow some strict rules. First, it is not allowed to touch any hardware. Instead, it is only allowed to install protocol instances onto its own Image Handle. A driver
that follows the UEFI Driver Model is required to install an instance of the Driver Binding Protocol onto its own Image Handle. It may optionally install the Driver Diagnostics Protocol or the Component Name Protocol. In addition, if a driver wishes to be unloadable it may optionally update the Loaded Image Protocol to provide its own Unload() function. Finally, if a driver needs to perform any special operations when the Boot Service EFI_BOOT_SERVICES is called (see Services — Boot Services), the driver may optionally create an event with a notification function that is triggered when the Boot Service ExitBootServices() is called. An Image Handle that contains a Driver Binding Protocol instance is known as a Driver Image Handle. Figure 14-8 shows a possible configuration for the Image Handle from Figure 14-7 after the Boot Service StartImage() has been called.

Figure 14-8 PCI Driver Image Handle

14.3.1.1 Driver Diagnostics Protocol

If a PCI Bus Driver or a PCI Device Driver requires diagnostics, then an EFI_DRIVER_DIAGNOSTICS2_PROTOCOL must be installed on the image handle in the entry point for the driver. This protocol contains functions to perform diagnostics on a controller. The EFI_DRIVER_DIAGNOSTICS2_PROTOCOL is not allowed to interact with the user. Instead, it must return status information through a buffer. The functions of this protocol will be invoked by a platform management utility.

14.3.1.2 Component Name Protocol

Both a PCI Bus Driver and a PCI Device Driver are able to produce user readable names for the PCI drivers and/or the set of PCI controllers that the PCI drivers are managing. This is accomplished by installing an instance of the EFI_COMPONENT_NAME2_PROTOCOL on the image handle of the driver. This protocol can produce driver and controller names in the form of a string in one of several languages. This protocol
can be used by a platform management utility to display user readable names for the drivers and controllers present in a system. Please see the EFI Driver Model Specification for details on the EFI_COMPONENT_NAME2_PROTOCOL.

14.3.1.3 Driver Family Override Protocol

If a PCI Bus Driver or PCI Device Driver always wants the PCI driver delivered in a PCI Option ROM to manage the PCI controller associated with the PCI Option ROM, then the Driver Family Override Protocol must not be produced.

If a PCI Bus Driver or PCI Device Driver always wants the PCI driver with the highest Version value in the Driver Binding Protocol to manage all the PCI Controllers in the same family of PCI controllers, then the Driver Family Override Protocol must be produced on the same handle as the Driver Binding Protocol.

14.3.2 PCI Bus Drivers

A PCI Bus Driver manages PCI Host Bus Controllers that can contain one or more PCI Root Bridges. Figure 14-9 shows an example of a desktop system that has one PCI Host Bus Controller with one PCI Root Bridge.

![Diagram of PCI Bus Components]

Figure 14-9 PCI Host Bus Controller

The PCI Host Bus Controller in Figure 14-9 is abstracted in software with the PCI Root Bridge I/O Protocol. A PCI Bus Driver will manage handles that contain this protocol. Figure 14-10 shows an example device handle for a PCI Host Bus Controller. It contains a Device Path Protocol instance and a PCI Root Bridge I/O Protocol Instance.
14.3.2.1 Driver Binding Protocol for PCI Bus Drivers

The Driver Binding Protocol contains three services. These are `Supported()`, `Start()`, and `Stop()`. `Supported()` tests to see if the PCI Bus Driver can manage a device handle. A PCI Bus Driver can only manage device handles that contain the Device Path Protocol and the PCI Root Bridge I/O Protocol, so a PCI Bus Driver must look for these two protocols on the device handle that is being tested.

The `Start()` function tells the PCI Bus Driver to start managing a device handle. The device handle should support the protocols shown in Figure 14-10. The PCI Root Bridge I/O Protocols provides access to the PCI I/O, PCI Memory, PCI Prefetchable Memory, and PCI DMA functions. The PCI Controllers behind a PCI Root Bridge may exist on one or more PCI Buses. The standard mechanism for expanding the number of PCI Buses on a single PCI Root Bridge is to use PCI to PCI Bridges. Once a PCI Enumerator configures these bridges, they are invisible to software. As a result, the PCI Bus Driver flattens the PCI Bus hierarchy when it starts managing a device handle that represents a PCI Host Controller. Figure 14-11 shows the physical tree structure for a set of PCI Device denoted by A, B, C, D, and E. Device A and C are PCI to PCI Bridges.
Figure 14-11 Physical PCI Bus Structure

Figure 14-12 shows the tree structure generated by a PCI Bus Driver before and after \texttt{Start()} is called. This is a logical view of set of PCI controller, and not a physical view. The physical tree is flattened, so any PCI to PCI bridge devices are invisible. In this example, the PCI Bus Driver finds the five child PCI Controllers on the PCI Bus from Figure 14-11. A device handle is created for every PCI Controller including all the PCI to PCI Bridges. The arrow with the dashed line coming into the PCI Host Bus Controller represents a link to the PCI Host Bus Controller’s parent. If the PCI Host Bus Controller is a Root Bus Controller, then it will not have a parent. The PCI Driver Model does not require that a PCI Host Bus Controller be a Root Bus Controller. A PCI Host Bus Controller can be present at any location in the tree, and the PCI Bus Driver should be able to manage the PCI Host Bus Controller.
The PCI Bus Driver has the option of creating all of its children in one call to \texttt{Start()}, or spreading it across several calls to \texttt{Start()}. In general, if it is possible to design a bus driver to create one child at a time, it should do so to support the rapid boot capability in the UEFI Driver Model. Each of the child device handles created in \texttt{Start()} must contain a Device Path Protocol instance, a PCI I/O protocol instance, and optionally a Bus Specific Driver Override Protocol instance. The PCI I/O Protocol is described in Section 14.4. The format of device paths for PCI Controllers is described in Section 2.6, and details on the Bus Specific Driver Override Protocol can be found in the \textit{EFI Driver Model Specification}. Figure 14-13 shows an example child device handle that is created by a PCI Bus Driver for a PCI Controller.

A PCI Bus Driver must perform several steps to manage a PCI Host Bus Controller, as follows:

- Initialize the PCI Host Bus Controller.
• If the PCI buses have not been initialized by a previous agent, perform PCI Enumeration on all the PCI Root Bridges that the PCI Host Bus Controller contains. This involves assigning a PCI bus number, allocating PCI I/O resources, PCI Memory resources, and PCI Prefetchable Memory resources.

• Discover all the PCI Controllers on all the PCI Root Bridges. If a PCI Controller is a PCI to PCI Bridge, then the I/O and Memory bits in the Control register of the PCI Configuration Header should be placed in the enabled state. The Bus Master bit in the Control Register may be enabled by default or enabled or disabled based on the needs of downstream devices for DMA access during the boot process. The PCI Bus Driver should disable the I/O, Memory, and Bus Master bits for PCI Controllers that respond to legacy ISA resources (e.g. VGA). It is a PCI Device Driver’s responsibility to enable the I/O, Memory, and Bus Master bits (if they are not already enabled by the PCI bus driver) of the Control register as required with a call to the `Attributes()` service when the PCI Device Driver is started. A similar call to the `Attributes()` service should be made when the PCI Device Driver is stopped to restore original `Attributes()` state, including the I/O, Memory, and Bus Master bits of the Control register.

• Create a device handle for each PCI Controller found. If a request is being made to start only one PCI Controller, then only create one device handle.

• Install a Device Path Protocol instance and a PCI I/O Protocol instance on the device handle created for each PCI Controller.

• If the PCI Controller has a PCI Option ROM, then allocate a memory buffer that is the same size as the PCI Option ROM, and copy the PCI Option ROM contents to the memory buffer.

• If the PCI Option ROM contains any UEFI drivers, then attach a Bus Specific Driver Override Protocol to the device handle of the PCI Controller that is associated with the PCI Option ROM.

The `Stop()` function tells the PCI Bus Driver to stop managing a PCI Host Bus Controller. The `Stop()` function can destroy one or more of the device handles that were created on a previous call to `Start()`. If all of the child device handles have been destroyed, then `Stop()` will place the PCI Host Bus Controller in a quiescent state. The functionality of `Stop()` mirrors `Start()`, as follows:

1. Complete all outstanding transactions to the PCI Host Bus Controller.
2. If the PCI Host Bus Controller is being stopped, then place it in a quiescent state.
3. If one or more child handles are being destroyed, then:
   a. Uninstall all the protocols from the device handles for the PCI Controllers found in `Start()`.
   b. Free any memory buffers allocated for PCI Option ROMs.
   c. Destroy the device handles for the PCI controllers created in `Start()`.

14.3.2.2 PCI Enumeration

The PCI Enumeration process is a platform-specific operation that depends on the properties of the chipset that produces the PCI bus. As a result, details on PCI Enumeration are outside the scope of this document. A PCI Bus Driver requires that PCI Enumeration has been performed, so it either needs to have been done prior to the PCI Bus Driver starting, or it must be part of the PCI Bus Driver’s implementation.
14.3.3 PCI Device Drivers

PCI Device Drivers manage PCI Controllers. Device handles for PCI Controllers are created by PCI Bus Drivers. A PCI Device Driver is not allowed to create any new device handles. Instead, it attaches protocol instance to the device handle of the PCI Controller. These protocol instances are I/O abstractions that allow the PCI Controller to be used in the preboot environment. The most common I/O abstractions are used to boot an EFI compliant OS.

14.3.3.1 Driver Binding Protocol for PCI Device Drivers

The Driver Binding Protocol contains three services. These are Supported(), Start(), and Stop(). Supported() tests to see if the PCI Device Driver can manage a device handle. A PCI Device Driver can only manage device handles that contain the Device Path Protocol and the PCI I/O Protocol, so a PCI Device Driver must look for these two protocols on the device handle that is being tested. In addition, it needs to check to see if the device handle represents a PCI Controller that the PCI Device Driver knows how to manage. This is typically done by using the services of the PCI I/O Protocol to read the PCI Configuration Header for the PCI Controller, and looking at the VendorId, DeviceId, and SubsystemId fields.

The Start() function tells the PCI Device Driver to start managing a PCI Controller. A PCI Device Driver is not allowed to create any new device handles. Instead, it installs one or more addition protocol instances on the device handle for the PCI Controller. A PCI Device Driver is not allowed to modify the resources allocated to a PCI Controller. These resource allocations are owned by the PCI Bus Driver or some other firmware component that initialized the PCI Bus prior to the execution of the PCI Bus Driver. This means that the PCI BARs (Base Address Registers) and the configuration of any PCI to PCI bridge controllers must not be modified by a PCI Device Driver. A PCI Bus Driver will leave a PCI Device in a disabled safe initial state. A PCI Device Driver should save the original Attributes() state. It is a PCI Device Driver’s responsibility to call Attributes() to enable the I/O, Memory, and Bus Master decodes if they are not already enabled by the PCI bus driver.

The Stop() function mirrors the Start() function, so the Stop() function completes any outstanding transactions to the PCI Controller and removes the protocol interfaces that were installed in Start(). Figure 14-14 shows the device handle for a PCI Controller before and after Start() is called. In this example, a PCI Device Driver is adding the Block I/O Protocol to the device handle for the PCI Controller. It is also a PCI Device Driver’s responsibility to restore original Attributes() state, including the I/O, Memory, and Bus Master decodes by calling Attributes().
This section provides a detailed description of the **EFI_PCI_IO_PROTOCOL**. This protocol is used by code, typically drivers, running in the EFI boot services environment to access memory and I/O on a PCI controller. In particular, functions for managing devices on PCI buses are defined here.

The interfaces provided in the **EFI_PCI_IO_PROTOCOL** are for performing basic operations to memory, I/O, and PCI configuration space. The system provides abstracted access to basic system resources to allow a driver to have a programmatic method to access these basic system resources. The main goal of this protocol is to provide an abstraction that simplifies the writing of device drivers for PCI devices. This goal is accomplished by providing the following features:

- A driver model that does not require the driver to search the PCI busses for devices to manage. Instead, drivers are provided the location of the device to manage or have the capability to be notified when a PCI controller is discovered.
- A device driver model that abstracts the I/O addresses, Memory addresses, and PCI Configuration addresses from the PCI device driver. Instead, BAR (Base Address Register) relative addressing is used for I/O and Memory accesses, and device relative addressing is used for PCI Configuration accesses. The BAR relative addressing is specified in the PCI I/O services as a BAR index. A PCI controller may contain a combination of 32-bit and 64-bit BARs. The BAR index represents the logical BAR number in the standard PCI configuration header starting...
from the first BAR. The BAR index does not represent an offset into the standard PCI
Configuration Header because those offsets will vary depending on the combination and order
of 32-bit and 64-bit BARs.

- The Device Path for the PCI device can be obtained from the same device handle that the
  EFI_PCI_IO_PROTOCOL resides.
- The PCI Segment, PCI Bus Number, PCI Device Number, and PCI Function Number of the PCI
device if they are required. The general idea is to abstract these details away from the PCI
device driver. However, if these details are required, then they are available.
- Details on any nonstandard address decoding that is not covered by the PCI device's Base
  Address Registers.
- Access to the PCI Root Bridge I/O Protocol for the PCI Host Bus for which the PCI device is a
  member.
- A copy of the PCI Option ROM if it is present in system memory.
- Functions to perform bus mastering DMA. This includes both packet based DMA and common
  buffer DMA.

**EFI_PCI_IO_PROTOCOL**

**Summary**

Provides the basic Memory, I/O, PCI configuration, and DMA interfaces that a driver uses to access its PCI controller.
GUID
#define EFI_PCI_IO_PROTOCOL_GUID
{0x4cf5b200,0x68b8,0x4ca5,
{0x9e,0xec,0xb2,0x3e,0x3f,0x50,0x02,0x9a}

Protocol Interface Structure

typedef struct _EFI_PCI_IO_PROTOCOL {
  EFI_PCI_IO_PROTOCOL_POLL_MEM      PollMem;
  EFI_PCI_IO_PROTOCOL_POLL_MEM      PollIo;
  EFI_PCI_IO_PROTOCOL_ACCESS       Mem;
  EFI_PCI_IO_PROTOCOL_ACCESS       Io;
  EFI_PCI_IO_PROTOCOL_CONFIG_ACCESS Pci;
  EFI_PCI_IO_PROTOCOL_COPY_MEM     CopyMem;
  EFI_PCI_IO_PROTOCOL_MAP          Map;
  EFI_PCI_IO_PROTOCOL_UNMAP        Unmap;
  EFI_PCI_IO_PROTOCOL_ALLOCATE_BUFFER AllocateBuffer;
  EFI_PCI_IO_PROTOCOL_FREE_BUFFER  FreeBuffer;
  EFI_PCI_IO_PROTOCOL_FLUSH        Flush;
  EFI_PCI_IO_PROTOCOL_GET_LOCATION GetLocation;
  EFI_PCI_IO_PROTOCOL_ATTRIBUTES   Attributes;
  EFI_PCI_IO_PROTOCOL_GET_BAR_ATTRIBUTES GetBarAttributes;
  EFI_PCI_IO_PROTOCOL_SET_BAR_ATTRIBUTES SetBarAttributes;
  UINT64 RomSize;
  VOID *RomImage;
} EFI_PCI_IO_PROTOCOL;

Parameters

PollMem
Polls an address in PCI memory space until an exit condition is met, or a timeout occurs. See the PollMem() function description.

PollIo
Polls an address in PCI I/O space until an exit condition is met, or a timeout occurs. See the PollIo() function description.

Mem.Read
Allows BAR relative reads to PCI memory space. See the Mem.Read() function description.

Mem.Write
Allows BAR relative writes to PCI memory space. See the Mem.Write() function description.

Io.Read
Allows BAR relative reads to PCI I/O space. See the Io.Read() function description.

Io.Write
Allows BAR relative writes to PCI I/O space. See the Io.Write() function description.

Pci.Read
Allows PCI controller relative reads to PCI configuration space. See the Pci.Read() function description.

Pci.Write
Allows PCI controller relative writes to PCI configuration space. See the Pci.Write() function description.

CopyMem
Allows one region of PCI memory space to be copied to another region of PCI memory space. See the CopyMem() function description.
<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Map</strong></td>
<td>Provides the PCI controller–specific address needed to access system memory for DMA. See the <code>Map()</code> function description.</td>
</tr>
<tr>
<td><strong>Unmap</strong></td>
<td>Releases any resources allocated by <code>Map()</code>. See the <code>Unmap()</code> function description.</td>
</tr>
<tr>
<td><strong>AllocateBuffer</strong></td>
<td>Allocates pages that are suitable for a common buffer mapping. See the <code>AllocateBuffer()</code> function description.</td>
</tr>
<tr>
<td><strong>FreeBuffer</strong></td>
<td>Frees pages that were allocated with <code>AllocateBuffer()</code>. See the <code>FreeBuffer()</code> function description.</td>
</tr>
<tr>
<td><strong>Flush</strong></td>
<td>Flushes all PCI posted write transactions to system memory. See the <code>Flush()</code> function description.</td>
</tr>
<tr>
<td><strong>GetLocation</strong></td>
<td>Retrieves this PCI controller’s current PCI bus number, device number, and function number. See the <code>GetLocation()</code> function description.</td>
</tr>
<tr>
<td><strong>Attributes</strong></td>
<td>Performs an operation on the attributes that this PCI controller supports. The operations include getting the set of supported attributes, retrieving the current attributes, setting the current attributes, enabling attributes, and disabling attributes. See the <code>Attributes()</code> function description.</td>
</tr>
<tr>
<td><strong>GetBarAttributes</strong></td>
<td>Gets the attributes that this PCI controller supports setting on a BAR using <code>SetBarAttributes()</code>, and retrieves the list of resource descriptors for a BAR. See the <code>GetBarAttributes()</code> function description.</td>
</tr>
<tr>
<td><strong>SetBarAttributes</strong></td>
<td>Sets the attributes for a range of a BAR on a PCI controller. See the <code>SetBarAttributes()</code> function description.</td>
</tr>
<tr>
<td><strong>RomSize</strong></td>
<td>The size, in bytes, of the ROM image.</td>
</tr>
<tr>
<td><strong>RomImage</strong></td>
<td>A pointer to the in memory copy of the ROM image. The PCI Bus Driver is responsible for allocating memory for the ROM image, and copying the contents of the ROM to memory. The contents of this buffer are either from the PCI option ROM that can be accessed through the ROM BAR of the PCI controller, or it is from a platform-specific location. The <code>Attributes()</code> function can be used to determine from which of these two sources the <code>RomImage</code> buffer was initialized.</td>
</tr>
</tbody>
</table>
Related Definitions

```
//******************************************************************************
// EFI_PCI_IO_PROTOCOL_WIDTH
//******************************************************************************
typedef enum {
  EfiPciIoWidthUint8,   
  EfiPciIoWidthUint16,  
  EfiPciIoWidthUint32,  
  EfiPciIoWidthUint64,  
  EfiPciIoWidthFifoUint8, 
  EfiPciIoWidthFifoUint16, 
  EfiPciIoWidthFifoUint32, 
  EfiPciIoWidthFifoUint64, 
  EfiPciIoWidthFillUint8, 
  EfiPciIoWidthFillUint16, 
  EfiPciIoWidthFillUint32, 
  EfiPciIoWidthFillUint64, 
  EfiPciIoWidthMaximum
} EFI_PCI_IO_PROTOCOL_WIDTH;
```

```
#define EFI_PCI_IO_PASS_THROUGH_BAR  0xff
```

```
//******************************************************************************
// EFI_PCI_IO_PROTOCOL_POLL_IO_MEM
//******************************************************************************
typedef
EFI_STATUS
(EFIAPICALLCONVENTION EFI_PCI_IO_PROTOCOL_POLL_IO_MEM) (    
  IN EFI_PCI_IO_PROTOCOL *This,  
  IN EFI_PCI_IO_PROTOCOL_WIDTH Width,  
  IN UINT8                BarIndex,  
  IN UINT64               Offset,  
  IN UINT64               Mask,  
  IN UINT64               Value,  
  IN UINT64               Delay,  
  OUT UINT64              *Result
);
```

```
//******************************************************************************
// EFI_PCI_IO_PROTOCOL_IO_MEM
//******************************************************************************
typedef
EFI_STATUS
(EFIAPICALLCONVENTION EFI_PCI_IO_PROTOCOL_IO_MEM) (    
  IN EFI_PCI_IO_PROTOCOL *This,  
  IN EFI_PCI_IO_PROTOCOL_WIDTH Width,  
  IN UINT8                BarIndex,  
  IN UINT64               Offset,  
```
typedef struct {
    EFI_PCI_IO_PROTOCOL_IO_MEM Read;
    EFI_PCI_IO_PROTOCOL_IO_MEM Write;
} EFI_PCI_IO_PROTOCOL_ACCESS;

typedef
EFI_STATUS
(EFIAPI *EFI_PCI_IO_PROTOCOL_CONFIG) (    
    IN EFI_PCI_IO_PROTOCOL *This,
    IN EFI_PCI_IO_PROTOCOL_WIDTH Width,
    IN UINT32 Offset,
    IN UINTN Count,
    IN OUT VOID *Buffer
);

typedef struct {
    EFI_PCI_IO_PROTOCOL_CONFIG Read;
    EFI_PCI_IO_PROTOCOL_CONFIG Write;
} EFI_PCI_IO_PROTOCOL_CONFIG_ACCESS;

#define EFI_PCI_IO_ATTRIBUTE_ISA_MOTHERBOARD_IO 0x0001
#define EFI_PCI_IO_ATTRIBUTE_ISA_IO               0x0002
#define EFI_PCI_IO_ATTRIBUTE_VGA_PALETTE_IO       0x0004
#define EFI_PCI_IO_ATTRIBUTE_VGA_MEMORY           0x0008
#define EFI_PCI_IO_ATTRIBUTE_VGA_IO               0x0010
#define EFI_PCI_IO_ATTRIBUTE_IDE_PRIMARY_IO       0x0020
#define EFI_PCI_IO_ATTRIBUTE_IDE_SECONDARY_IO     0x0040
#define EFI_PCI_IO_ATTRIBUTE_MEMORY_WRITE_COMBINE 0x0080
#define EFI_PCI_IO_ATTRIBUTE_IO                   0x0100
#define EFI_PCI_IO_ATTRIBUTE_MEMORY               0x0200
#define EFI_PCI_IO_ATTRIBUTE_BUS_MASTER           0x0400
#define EFI_PCI_IO_ATTRIBUTE_MEMORY_CACHED        0x0800
```c
#define EFI_PCI_IO_ATTRIBUTE_MEMORY_DISABLE       0x1000
#define EFI_PCI_IO_ATTRIBUTE_EMBEDDED_DEVICE      0x2000
#define EFI_PCI_IO_ATTRIBUTE_EMBEDDED_ROM         0x4000
#define EFI_PCI_IO_ATTRIBUTE_DUAL_ADDRESS_CYCLE   0x8000
#define EFI_PCI_IO_ATTRIBUTE_ISA_IO_16            0x10000
#define EFI_PCI_IO_ATTRIBUTE_VGA_PALETTE_IO_16    0x20000
#define EFI_PCI_IO_ATTRIBUTE_VGA_IO_16            0x40000

EFI_PCI_IO_ATTRIBUTE_ISA_IO_16
If this bit is set, then the PCI I/O cycles between 0x100 and 0x3FF are forwarded to
the PCI controller using a 16-bit address decoder on address bits 0..15. Address bits
16..31 must be zero. This bit is used to forward I/O cycles for legacy ISA devices. If
this bit is set, then the PCI Host Bus Controller and all the PCI to PCI bridges between
the PCI Host Bus Controller and the PCI Controller are configured to forward these
PCI I/O cycles. This bit may not be combined with
EFI_PCI_IO_ATTRIBUTE_ISA_IO.

EFI_PCI_IO_ATTRIBUTE_VGA_PALETTE_IO_16
If this bit is set, then the PCI I/O write cycles for 0x3C6, 0x3C8, and 0x3C9 are
forwarded to the PCI controller using a 16-bit address decoder on address bits 0..15.
Address bits 16..31 must be zero. This bit is used to forward I/O write cycles to the
VGA palette registers on a PCI controller. If this bit is set, then the PCI Host Bus
Controller and all the PCI to PCI bridges between the PCI Host Bus Controller and the
PCI Controller are configured to forward these PCI I/O cycles. This bit may not be
combined with EFI_PCI_IO_ATTRIBUTE_VGA_IO or
EFI_PCI_IO_ATTRIBUTE_VGA_PALETTE_IO.

EFI_PCI_IO_ATTRIBUTE_VGA_IO_16
If this bit is set, then the PCI I/O cycles in the ranges 0x3B0–0x3BB and 0x3C0–0x3DF
are forwarded to the PCI controller using a 16-bit address decoder on address bits
0..15. Address bits 16..31 must be zero. This bit is used to forward I/O cycles for a
VGA controller to a PCI controller. If this bit is set, then the PCI Host Bus Controller
and all the PCI to PCI bridges between the PCI Host Bus Controller and the PCI
Controller are configured to forward these PCI I/O cycles. This bit may not be
combined with EFI_PCI_IO_ATTRIBUTE_VGA_IO or
EFI_PCI_IO_ATTRIBUTE_VGA_PALETTE_IO. Because
EFI_PCI_IO_ATTRIBUTE_VGA_PALETTE_IO_16 also includes the I/O range described by
EFI_PCI_IO_ATTRIBUTE_VGA_PALETTE_IO_16, the
EFI_PCI_IO_ATTRIBUTE_VGA_PALETTE_IO_16 bit is ignored if
EFI_PCI_IO_ATTRIBUTE_VGA_IO_16 is set.

EFI_PCI_IO_ATTRIBUTE_ISA_MOTHERBOARD_IO
If this bit is set, then the PCI I/O cycles between 0x00000000 and 0x000000FF are
forwarded to the PCI controller. This bit is used to forward I/O cycles for ISA
motherboard devices. If this bit is set, then the PCI Host Bus Controller and all the PCI
to PCI bridges between the PCI Host Bus Controller and the PCI Controller are
configured to forward these PCI I/O cycles.
EFI_PCI_IO_ATTRIBUTE_ISA_IO
If this bit is set, then the PCI I/O cycles between 0x100 and 0x3FF are forwarded to
the PCI controller using a 10-bit address decoder on address bits 0..9. Address bits
10..15 are not decoded, and address bits 16..31 must be zero. This bit is used to
forward I/O cycles for legacy ISA devices. If this bit is set, then the PCI Host Bus
Controller and all the PCI to PCI bridges between the PCI Host Bus Controller and the
PCI Controller are configured to forward these PCI I/O cycles.

EFI_PCI_IO_ATTRIBUTE_VGA_PALETTE_IO
If this bit is set, then the PCI I/O write cycles for 0x3C6, 0x3C8, and 0x3C9 are
forwarded to the PCI controller using a 10-bit address decoder on address bits 0..9.
Address bits 10..15 are not decoded, and address bits 16..31 must be zero. This bit is
used to forward I/O write cycles to the VGA palette registers on a PCI controller. If
this bit is set, then the PCI Host Bus Controller and all the PCI to PCI bridges between
the PCI Host Bus Controller and the PCI Controller are configured to forward these
PCI I/O cycles.

EFI_PCI_IO_ATTRIBUTE_VGA_MEMORY
If this bit is set, then the PCI memory cycles between 0xA0000 and 0xBFFFF are
forwarded to the PCI controller. This bit is used to forward memory cycles for a VGA
frame buffer on a PCI controller. If this bit is set, then the PCI Host Bus Controller and
all the PCI to PCI bridges between the PCI Host Bus Controller and the PCI Controller
are configured to forward these PCI Memory cycles.

EFI_PCI_IO_ATTRIBUTE_VGA_IO
If this bit is set, then the PCI I/O cycles in the ranges 0x3B0-0x3BB and 0x3C0-0x3DF
are forwarded to the PCI controller using a 10-bit address decoder on address bits
0..9. Address bits 10..15 are not decoded, and the address bits 16..31 must be zero.
This bit is used to forward I/O cycles for a VGA controller to a PCI controller. If this bit
is set, then the PCI Host Bus Controller and all the PCI to PCI bridges between the PCI
Host Bus Controller and the PCI Controller are configured to forward these PCI I/O
cycles. Since EFI_PCI_IO_ATTRIBUTE_VGA_IO also includes the I/O range described
by EFI_PCI_IO_ATTRIBUTE_VGA_PALETTE_IO, the
EFI_PCI_IO_ATTRIBUTE_VGA_PALETTE_IO bit is ignored if
EFI_PCI_IO_ATTRIBUTE_VGA_IO is set.

EFI_PCI_IO_ATTRIBUTE_IDE_PRIMARY_IO
If this bit is set, then the PCI I/O cycles in the ranges 0x1F0-0x1F7 and 0x3F6-0x3F7
are forwarded to a PCI controller using a 16-bit address decoder on address bits
0..15. Address bits 16..31 must be zero. This bit is used to forward I/O cycles for a
Primary IDE controller to a PCI controller. If this bit is set, then the PCI Host Bus
Controller and all the PCI to PCI bridges between the PCI Host Bus Controller and the
PCI Controller are configured to forward these PCI I/O cycles.

EFI_PCI_IO_ATTRIBUTE_IDE_SECONDARY_IO
If this bit is set, then the PCI I/O cycles in the ranges 0x170-0x177 and 0x376-0x377
are forwarded to a PCI controller using a 16-bit address decoder on address bits
0..15. Address bits 16..31 must be zero. This bit is used to forward I/O cycles for a
Secondary IDE controller to a PCI controller. If this bit is set, then the PCI Host Bus
Controller and all the PCI to PCI bridges between the PCI Host Bus Controller and the PCI Controller are configured to forward these PCI I/O cycles.

**EFI_PCI_IO_ATTRIBUTE_MEMORY_WRITE_COMBINE**

If this bit is set, then this platform supports changing the attributes of a PCI memory range so that the memory range is accessed in a write combining mode. This bit is used to improve the write performance to a memory buffer on a PCI controller. By default, PCI memory ranges are not accessed in a write combining mode.

**EFI_PCI_IO_ATTRIBUTE_MEMORY_CACHED**

If this bit is set, then this platform supports changing the attributes of a PCI memory range so that the memory range is accessed in a cached mode. By default, PCI memory ranges are accessed noncached.

**EFI_PCI_IO_ATTRIBUTE_IO**

If this bit is set, then the PCI device will decode the PCI I/O cycles that the device is configured to decode.

**EFI_PCI_IO_ATTRIBUTE_MEMORY**

If this bit is set, then the PCI device will decode the PCI Memory cycles that the device is configured to decode.

**EFI_PCI_IO_ATTRIBUTE_BUS_MASTER**

If this bit is set, then the PCI device is allowed to act as a bus master on the PCI bus.

**EFI_PCI_IO_ATTRIBUTE_MEMORY_DISABLE**

If this bit is set, then this platform supports changing the attributes of a PCI memory range so that the memory range is disabled, and can no longer be accessed. By default, all PCI memory ranges are enabled.

**EFI_PCI_IO_ATTRIBUTE_EMBEDDED_DEVICE**

If this bit is set, then the PCI controller is an embedded device that is typically a component on the system board. If this bit is clear, then this PCI controller is part of an adapter that is populating one of the systems PCI slots.

**EFI_PCI_IO_ATTRIBUTE_EMBEDDED_ROM**

If this bit is set, then the PCI option ROM described by the `RomImage` and `RomSize` fields is not from ROM BAR of the PCI controller. If this bit is clear, then the `RomImage` and `RomSize` fields were initialized based on the PCI option ROM found through the ROM BAR of the PCI controller.

**EFI_PCI_IO_ATTRIBUTE_DUAL_ADDRESS_CYCLE**

If this bit is set, then the PCI controller is capable of producing PCI Dual Address Cycles, so it is able to access a 64-bit address space. If this bit is not set, then the PCI controller is not capable of producing PCI Dual Address Cycles, so it is only able to access a 32-bit address space.

If this bit is set, then the PCI Host Bus Controller and all the PCI to PCI bridges between the PCI Host Bus Controller and the PCI Controller are capable of producing PCI Dual Address Cycles. If any of them is not capable of producing PCI Dual Address Cycles, attempt to perform Set or Enable operation using `Attributes()` function with this bit set will fail with the `EFI_UNSUPPORTED` error code.
typedef enum {
    EfiPciIoOperationBusMasterRead,
    EfiPciIoOperationBusMasterWrite,
    EfiPciIoOperationBusMasterCommonBuffer,
    EfiPciIoOperationMaximum
} EFI_PCI_IO_PROTOCOL_OPERATION;

EfiPciIoOperationBusMasterRead
    A read operation from system memory by a bus master.
EfiPciIoOperationBusMasterWrite
    A write operation to system memory by a bus master.
EfiPciIoOperationBusMasterCommonBuffer
    Provides both read and write access to system memory by both the processor and a bus master. The buffer is coherent from both the processor’s and the bus master’s point of view.

Description
The EFI_PCI_IO_PROTOCOL provides the basic Memory, I/O, PCI configuration, and DMA interfaces that are used to abstract accesses to PCI controllers. There is one EFI_PCI_IO_PROTOCOL instance for each PCI controller on a PCI bus. A device driver that wishes to manage a PCI controller in a system will have to retrieve the EFI_PCI_IO_PROTOCOL instance that is associated with the PCI controller. A device handle for a PCI controller will minimally contain an EFI_DEVICE_PATH_PROTOCOL instance and an EFI_PCI_IO_PROTOCOL instance.

Bus mastering PCI controllers can use the DMA services for DMA operations. There are three basic types of bus mastering DMA that is supported by this protocol. These are DMA reads by a bus master, DMA writes by a bus master, and common buffer DMA. The DMA read and write operations may need to be broken into smaller chunks. The caller of Map() must pay attention to the number of bytes that were mapped, and if required, loop until the entire buffer has been transferred. The following is a list of the different bus mastering DMA operations that are supported, and the sequence of EFI_PCI_IO_PROTOCOL interfaces that are used for each DMA operation type.

DMA Bus Master Read Operation
Call Map() for EfiPciIoOperationBusMasterRead.
Program the DMA Bus Master with the DeviceAddress returned by Map().
Start the DMA Bus Master.
Wait for DMA Bus Master to complete the read operation.
Call Unmap().

DMA Bus Master Write Operation
Call Map() for EfiPciOperationBusMasterWrite.
Program the DMA Bus Master with the DeviceAddress returned by Map().
Start the DMA Bus Master.
Wait for DMA Bus Master to complete the write operation.
Perform a PCI controller specific read transaction to flush all PCI write buffers (See PCI Specification Section 3.2.5.2).
Call Flush().
Call Unmap().

DMA Bus Master Common Buffer Operation
Call AllocateBuffer() to allocate a common buffer.
Call Map() for EfiPciloOperationBusMasterCommonBuffer.
Program the DMA Bus Master with the DeviceAddress returned by Map().
The common buffer can now be accessed equally by the processor and the DMA bus master.
Call Unmap().
Call FreeBuffer().

**EFI_PCI_IO_PROTOCOL.PollMem()**

**Summary**
Reads from the memory space of a PCI controller. Returns when either the polling exit criteria is satisfied or after a defined duration.

**Prototype**
```
typedef
EFI_STATUS
(EFIAPI *EFI_PCI_IO_PROTOCOL_POLL_IO_MEM) (
    IN EFI_PCI_IO_PROTOCOL *This,
    IN EFI_PCI_IO_PROTOCOL_WIDTH Width,
    IN UINT8 BarIndex,
    IN UINT64 Offset,
    IN UINT64 Mask,
    IN UINT64 Value,
    IN UINT64 Delay,
    OUT UINT64 *Result

);
```

**Parameters**
- **This**
  A pointer to the EFI_PCI_IO_PROTOCOL instance. Type **EFI_PCI_IO_PROTOCOL** is defined in Section 14.4.
- **Width**
  Signifies the width of the memory operations. Type **EFI_PCI_IO_PROTOCOL_WIDTH** is defined in Section 14.4.
**BarIndex**

The BAR index of the standard PCI Configuration header to use as the base address for the memory operation to perform. This allows all drivers to use BAR relative addressing. The legal range for this field is 0..5. However, the value `EFI_PCI_IO_PASS_THROUGH_BAR` can be used to bypass the BAR relative addressing and pass `Offset` to the PCI Root Bridge I/O Protocol unchanged. Type `EFI_PCI_IO_PASS_THROUGH_BAR` is defined in Section 14.4.

**Offset**

The offset within the selected BAR to start the memory operation.

**Mask**

Mask used for the polling criteria. Bytes above `Width` in `Mask` are ignored. The bits in the bytes below `Width` which are zero in `Mask` are ignored when polling the memory address.

**Value**

The comparison value used for the polling exit criteria.

**Delay**

The number of 100 ns units to poll. Note that timer available may be of poorer granularity.

**Result**

Pointer to the last value read from the memory location.

**Description**

This function provides a standard way to poll a PCI memory location. A PCI memory read operation is performed at the PCI memory address specified by `BarIndex` and `Offset` for the width specified by `Width`. The result of this PCI memory read operation is stored in `Result`. This PCI memory read operation is repeated until either a timeout of `Delay` 100 ns units has expired, or `(Result & Mask)` is equal to `Value`.

This function will always perform at least one memory access no matter how small `Delay` may be. If `Delay` is 0, then `Result` will be returned with a status of `EFI_SUCCESS` even if `Result` does not match the exit criteria. If `Delay` expires, then `EFI_TIMEOUT` is returned.

If `Width` is not `EfiPciIoWidthUint8`, `EfiPciIoWidthUint16`, `EfiPciIoWidthUint32`, or `EfiPciIoWidthUint64`, then `EFI_INVALID_PARAMETER` is returned.

The memory operations are carried out exactly as requested. The caller is responsible for satisfying any alignment and memory width restrictions that a PCI controller on a platform might require. For example on some platforms, width requests of `EfiPciIoWidthUint64` do not work.

All the PCI transactions generated by this function are guaranteed to be completed before this function returns. However, if the memory mapped I/O region being accessed by this function has the `EFI_PCI_IO_ATTRIBUTE_MEMORY_CACHED` attribute set, then the transactions will follow the ordering rules defined by the processor architecture.
Status Codes Returned

<table>
<thead>
<tr>
<th>Code</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>EFI_SUCCESS</td>
<td>The last data returned from the access matched the poll exit criteria.</td>
</tr>
<tr>
<td>EFI_INVALID_PARAMETER</td>
<td>Width is invalid.</td>
</tr>
<tr>
<td>EFI_INVALID_PARAMETER</td>
<td>Result is NULL.</td>
</tr>
<tr>
<td>EFI_UNSUPPORTED</td>
<td>BarIndex not valid for this PCI controller.</td>
</tr>
<tr>
<td>EFI_UNSUPPORTED</td>
<td>Offset is not valid for the BarIndex of this PCI controller.</td>
</tr>
<tr>
<td>EFI_TIMEOUT</td>
<td>Delay expired before a match occurred.</td>
</tr>
<tr>
<td>EFI_OUT_OF_RESOURCES</td>
<td>The request could not be completed due to a lack of resources.</td>
</tr>
</tbody>
</table>

**EFI_PCI_IO_PROTOCOL.PollIo()**

**Summary**

Reads from the I/O space of a PCI controller. Returns when either the polling exit criteria is satisfied or after a defined duration.

**Prototype**

```c
typedef EFI_STATUS
(EFIAPI *EFI_PCI_IO_PROTOCOL_POLL_IO_MEM) (  
    IN EFI_PCI_IO_PROTOCOL *This,  
    IN EFI_PCI_IO_PROTOCOL_WIDTH Width,  
    IN UINT8 BarIndex,  
    IN UINT64 Offset,  
    IN UINT64 Mask,  
    IN UINT64 Value,  
    IN UINT64 Delay,  
    OUT UINT64 *Result
);
```

**Parameters**

- **This**
  - A pointer to the EFI_PCI_IO_PROTOCOL instance. Type EFI_PCI_IO_PROTOCOL is defined in Section 14.4.

- **Width**
  - Signifies the width of the I/O operations. Type EFI_PCI_IO_PROTOCOL_WIDTH is defined in Section 14.4.

- **BarIndex**
  - The BAR index of the standard PCI Configuration header to use as the base address for the I/O operation to perform. This allows all drivers to use BAR relative addressing. The legal range for this field is 0..5. However, the value EFI_PCI_IO_PASS_THROUGH_BAR can be used to bypass the BAR relative addressing and pass Offset to the PCI Root Bridge I/O Protocol unchanged. Type EFI_PCI_IO_PASS_THROUGH_BAR is defined in Section 14.4.

- **Offset**
  - The offset within the selected BAR to start the I/O operation.

- **Mask**
  - Mask used for the polling criteria. Bytes above Width in Mask are ignored. The bits in the bytes below Width which are zero in Mask are ignored when polling the I/O address.
Value

The comparison value used for the polling exit criteria.

Delay

The number of 100 ns units to poll. Note that timer available may be of poorer granularity.

Result

Pointer to the last value read from the memory location.

Description

This function provides a standard way to poll a PCI I/O location. A PCI I/O read operation is performed at the PCI I/O address specified by BarIndex and Offset for the width specified by Width. The result of this PCI I/O read operation is stored in Result. This PCI I/O read operation is repeated until either a timeout of Delay 100 ns units has expired, or (Result & Mask) is equal to Value.

This function will always perform at least one I/O access no matter how small Delay may be. If Delay is 0, then Result will be returned with a status of EFI_SUCCESS even if Result does not match the exit criteria. If Delay expires, then EFI_TIMEOUT is returned.

If Width is not EfiPciIoWidthUint8, EfiPciIoWidthUint16, EfiPciIoWidthUint32, or EfiPciIoWidthUint64, then EFI_INVALID_PARAMETER is returned.

The I/O operations are carried out exactly as requested. The caller is responsible satisfying any alignment and I/O width restrictions that the PCI controller on a platform might require. For example on some platforms, width requests of EfiPciIoWidthUint64 do not work.

All the PCI read transactions generated by this function are guaranteed to be completed before this function returns.

Status Codes Returned

<table>
<thead>
<tr>
<th>Status Code</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>EFI_SUCCESS</td>
<td>The last data returned from the access matched the poll exit criteria.</td>
</tr>
<tr>
<td>EFI_INVALID_PARAMETER</td>
<td>Width is invalid.</td>
</tr>
<tr>
<td>EFI_INVALID_PARAMETER</td>
<td>Result is NULL.</td>
</tr>
<tr>
<td>EFI_UNSUPPORTED</td>
<td>BarIndex not valid for this PCI controller.</td>
</tr>
<tr>
<td>EFI_UNSUPPORTED</td>
<td>Offset is not valid for the PCI BAR specified by BarIndex.</td>
</tr>
<tr>
<td>EFI_TIMEOUT</td>
<td>Delay expired before a match occurred.</td>
</tr>
<tr>
<td>EFI_OUT_OF_RESOURCES</td>
<td>The request could not be completed due to a lack of resources.</td>
</tr>
</tbody>
</table>

EFI_PCI_IO_PROTOCOL.Mem.Read()

EFI_PCI_IO_PROTOCOL.Mem.Write()

Summary

Enable a PCI driver to access PCI controller registers in the PCI memory space.
Prototype

```c
typedef EFI_STATUS
  (EFIAPI *EFI_PCI_IO_PROTOCOL_MEM) (
    IN   EFI_PCI_IO_PROTOCOL   *This,
    IN   EFI_PCI_IO_PROTOCOL_WIDTH   Width,
    IN   UINT8   BarIndex,
    IN   UINT64   Offset,
    IN   UINTN   Count,
    IN OUT VOID   *Buffer
  );
```

Parameters

- **This**: A pointer to the `EFI_PCI_IO_PROTOCOL` instance. Type `EFI_PCI_IO_PROTOCOL` is defined in Section 14.4.
- **Width**: Signifies the width of the memory operations. Type `EFI_PCI_IO_PROTOCOL_WIDTH` is defined in Section 14.4.
- **BarIndex**: The BAR index of the standard PCI Configuration header to use as the base address for the memory operation to perform. This allows all drivers to use BAR relative addressing. The legal range for this field is 0..5. However, the value `EFI_PCI_IO_PASS_THROUGH_BAR` can be used to bypass the BAR relative addressing and pass `Offset` to the PCI Root Bridge I/O Protocol unchanged. Type `EFI_PCI_IO_PASS_THROUGH_BAR` is defined in Section 14.4.
- **Offset**: The offset within the selected BAR to start the memory operation.
- **Count**: The number of memory operations to perform. Bytes moved is `Width` size * `Count`, starting at `Offset`.
- **Buffer**: For read operations, the destination buffer to store the results. For write operations, the source buffer to write data from.

Description

The `Mem.Read()` and `Mem.Write()` functions enable a driver to access controller registers in the PCI memory space.

The I/O operations are carried out exactly as requested. The caller is responsible for any alignment and I/O width issues which the bus, device, platform, or type of I/O might require. For example on some platforms, width requests of `EfiPciIoWidthUint64` do not work.

If `Width` is `EfiPciIoWidthUint8`, `EfiPciIoWidthUint16`, `EfiPciIoWidthUint32`, or `EfiPciIoWidthUint64`, then both `Address` and `Buffer` are incremented for each of the `Count` operations performed.

If `Width` is `EfiPciIoWidthFifoUint8`, `EfiPciIoWidthFifoUint16`, `EfiPciIoWidthFifoUint32`, or `EfiPciIoWidthFifoUint64`, then only `Buffer` is incremented for each of the `Count` operations performed. The read or write operation is performed `Count` times on the same `Address`. 
If `Width` is `EfiPciIoWidthFillUint8`, `EfiPciIoWidthFillUint16`, `EfiPciIoWidthFillUint32`, or `EfiPciIoWidthFillUint64`, then only `Address` is incremented for each of the `Count` operations performed. The read or write operation is performed `Count` times from the first element of `Buffer`.

All the PCI transactions generated by this function are guaranteed to be completed before this function returns. All the PCI write transactions generated by this function will follow the write ordering and completion rules defined in the `PCI Specification`. However, if the memory-mapped I/O region being accessed by this function has the `EFI_PCI_IO_ATTRIBUTE_MEMORY_CACHED` attribute set, then the transactions will follow the ordering rules defined by the processor architecture.

### Status Codes Returned

<table>
<thead>
<tr>
<th>Status Code</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>EFI_SUCCESS</td>
<td>The data was read from or written to the PCI controller.</td>
</tr>
<tr>
<td>EFI_INVALID_PARAMETER</td>
<td><code>Width</code> is invalid.</td>
</tr>
<tr>
<td>EFI_INVALID_PARAMETER</td>
<td><code>Buffer</code> is <code>NULL</code>.</td>
</tr>
<tr>
<td>EFI_UNSUPPORTED</td>
<td><code>BarIndex</code> not valid for this PCI controller.</td>
</tr>
<tr>
<td>EFI_UNSUPPORTED</td>
<td>The address range specified by <code>Offset</code>, <code>Width</code>, and <code>Count</code> is not valid</td>
</tr>
<tr>
<td></td>
<td>for the PCI BAR specified by <code>BarIndex</code>.</td>
</tr>
<tr>
<td>EFI_OUT_OF_RESOURCES</td>
<td>The request could not be completed due to a lack of resources.</td>
</tr>
</tbody>
</table>

### EFI_PCI_IO_PROTOCOL.Io.Read()

### EFI_PCI_IO_PROTOCOL.Io.Write()

**Summary**

Enable a PCI driver to access PCI controller registers in the PCI I/O space.

**Prototype**

```c
typedef EFI_STATUS (EFIAPI *EFI_PCI_IO_PROTOCOL_MEM) (  
    IN  EFI_PCI_IO_PROTOCOL *This,  
    IN  EFI_PCI_IO_PROTOCOL_WIDTH Width,  
    IN  UINT8 BarIndex,  
    IN  UINT64 Offset,  
    IN  UINTN Count,  
    IN OUT VOID *Buffer  
);
```

**Parameters**

- `This` A pointer to the `EFI_PCI_IO_PROTOCOL` instance. Type `EFI_PCI_IO_PROTOCOL` is defined in Section 14.4.
- `Width` Signifies the width of the memory operations. Type `EFI_PCI_IO_PROTOCOL_WIDTH` is defined in Section 14.4.
- `BarIndex` The BAR index in the standard PCI Configuration header to use as the base address for the I/O operation to perform. This allows all drivers
Offset
The offset within the selected BAR to start the I/O operation.

Count
The number of I/O operations to perform. Bytes moved is Width size * Count, starting at Offset.

Buffer
For read operations, the destination buffer to store the results. For write operations, the source buffer to write data from.

Description
The Io.Read(), and Io.Write() functions enable a driver to access PCI controller registers in PCI I/O space.

The I/O operations are carried out exactly as requested. The caller is responsible for any alignment and I/O width issues which the bus, device, platform, or type of I/O might require. For example on some platforms, width requests of EfiPciIoWidthUint64 do not work.

If Width is EfiPciIoWidthUint8, EfiPciIoWidthUint16, EfiPciIoWidthUint32, or EfiPciIoWidthUint64, then both Address and Buffer are incremented for each of the Count operations performed.

If Width is EfiPciIoWidthFifoUint8, EfiPciIoWidthFifoUint16, EfiPciIoWidthFifoUint32, or EfiPciIoWidthFifoUint64, then only Buffer is incremented for each of the Count operations performed. The read or write operation is performed Count times on the same Address.

If Width is EfiPciIoWidthFillUint8, EfiPciIoWidthFillUint16, EfiPciIoWidthFillUint32, or EfiPciIoWidthFillUint64, then only Address is incremented for each of the Count operations performed. The read or write operation is performed Count times from the first element of Buffer.

All the PCI transactions generated by this function are guaranteed to be completed before this function returns.
Status Codes Returned

<table>
<thead>
<tr>
<th>Status Code</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>EFI_SUCCESS</td>
<td>The data was read from or written to the PCI controller.</td>
</tr>
<tr>
<td>EFI_INVALID_PARAMETER</td>
<td>Width is invalid.</td>
</tr>
<tr>
<td>EFI_INVALID_PARAMETER</td>
<td>Buffer is NULL.</td>
</tr>
<tr>
<td>EFI_UNSUPPORTED</td>
<td>BarIndex not valid for this PCI controller.</td>
</tr>
<tr>
<td>EFI_UNSUPPORTED</td>
<td>The address range specified by Offset, Width, and Count is not valid</td>
</tr>
<tr>
<td></td>
<td>for the PCI BAR specified by BarIndex.</td>
</tr>
<tr>
<td>EFI_OUT_OF_RESOURCES</td>
<td>The request could not be completed due to a lack of resources.</td>
</tr>
</tbody>
</table>

**EFI_PCI_IO_PROTOCOL.Pci.Read()**

**EFI_PCI_IO_PROTOCOL.Pci.Write()**

**Summary**

Enable a PCI driver to access PCI controller registers in PCI configuration space.

**Prototype**

```c
typedef EFI_STATUS
    (EFIAPI *EFI_PCI_IO_PROTOCOL_CONFIG) (  
    IN   EFI_PCI_IO_PROTOCOL  *This,
    IN   EFI_PCI_IO_PROTOCOL_WIDTH Width,
    IN   UINT32              Offset,
    IN   UINTN               Count,
    IN OUT VOID              *Buffer
    );
```

**Parameters**

- **This**
  
  A pointer to the EFI_PCI_IO_PROTOCOL instance. Type EFI_PCI_IO_PROTOCOL is defined in Section 14.4.

- **Width**
  
  Signifies the width of the memory operations. Type EFI_PCI_IO_PROTOCOL_WIDTH is defined in Section 14.4.

- **Offset**
  
  The offset within the PCI configuration space for the PCI controller.

- **Count**
  
  The number of PCI configuration operations to perform. Bytes moved is Width size * Count, starting at Offset.

- **Buffer**
  
  For read operations, the destination buffer to store the results. For write operations, the source buffer to write data from.

**Description**

The **Pci.Read()** and **Pci.Write()** functions enable a driver to access PCI configuration registers for the PCI controller.

The PCI Configuration operations are carried out exactly as requested. The caller is responsible for any alignment and I/O width issues which the bus, device, platform, or type of I/O might require. For example on some platforms, width requests of EfiPciIoWidthUint64 do not work.
If `Width` is `EfiPciIoWidthUint8`, `EfiPciIoWidthUint16`, `EfiPciIoWidthUint32`, or `EfiPciIoWidthUint64`, then both `Address` and `Buffer` are incremented for each of the `Count` operations performed.

If `Width` is `EfiPciIoWidthFifoUint8`, `EfiPciIoWidthFifoUint16`, `EfiPciIoWidthFifoUint32`, or `EfiPciIoWidthFifoUint64`, then only `Buffer` is incremented for each of the `Count` operations performed. The read or write operation is performed `Count` times on the same `Address`.

If `Width` is `EfiPciIoWidthFillUint8`, `EfiPciIoWidthFillUint16`, `EfiPciIoWidthFillUint32`, or `EfiPciIoWidthFillUint64`, then only `Address` is incremented for each of the `Count` operations performed. The read or write operation is performed `Count` times from the first element of `Buffer`.

All the PCI transactions generated by this function are guaranteed to be completed before this function returns.

### Status Codes Returned

<table>
<thead>
<tr>
<th>Code</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>EFI_SUCCESS</td>
<td>The data was read from or written to the PCI controller.</td>
</tr>
<tr>
<td>EFI_INVALID_PARAMETER</td>
<td>Width is invalid.</td>
</tr>
<tr>
<td>EFI_INVALID_PARAMETER</td>
<td>Buffer is NULL.</td>
</tr>
<tr>
<td>EFI_UNSUPPORTED</td>
<td>The address range specified by <code>Offset</code>, <code>Width</code>, and <code>Count</code> is not valid for the PCI configuration header of the PCI controller.</td>
</tr>
<tr>
<td>EFI_OUT_OF_RESOURCES</td>
<td>The request could not be completed due to a lack of resources.</td>
</tr>
</tbody>
</table>

### EFI_PCI_IO_PROTOCOL.CopyMem()

#### Summary
Enables a PCI driver to copy one region of PCI memory space to another region of PCI memory space.

#### Prototype

```c
typedef EFI_STATUS
  (EFIAPI *EFI_PCI_IO_PROTOCOL_COPY_MEM) (  
    IN   EFI_PCI_IO_PROTOCOL   *This,       
    IN   EFI_PCI_IO_PROTOCOL_WIDTH Width,   
    IN   UINT8                DestBarIndex, 
    IN   UINT64               DestOffset,   
    IN   UINT8                SrcBarIndex,  
    IN   UINT64               SrcOffset,    
    IN   UINTN                Count);       

Parameters

*This* A pointer to the EFI_PCI_IO_PROTOCOL instance. Type `EFI_PCI_IO_PROTOCOL` is defined in Section 14.4.
Width  Signifies the width of the memory operations. Type EFI_PCI_IO_PROTOCOL_WIDTH is defined in Section 14.4.

DestBarIndex  The BAR index in the standard PCI Configuration header to use as the base address for the memory operation to perform. This allows all drivers to use BAR relative addressing. The legal range for this field is 0..5. However, the value EFI_PCI_IO_PASS_THROUGH_BAR can be used to bypass the BAR relative addressing and pass Offset to the PCI Root Bridge I/O Protocol unchanged. Type EFI_PCI_IO_PASS_THROUGH_BAR is defined in Section 14.4.

DestOffset  The destination offset within the BAR specified by DestBarIndex to start the memory writes for the copy operation. The caller is responsible for aligning the DestOffset if required.

SrcBarIndex  The BAR index in the standard PCI Configuration header to use as the base address for the memory operation to perform. This allows all drivers to use BAR relative addressing. The legal range for this field is 0..5. However, the value EFI_PCI_IO_PASS_THROUGH_BAR can be used to bypass the BAR relative addressing and pass Offset to the PCI Root Bridge I/O Protocol unchanged. Type EFI_PCI_IO_PASS_THROUGH_BAR is defined in Section 14.4.

SrcOffset  The source offset within the BAR specified by SrcBarIndex to start the memory reads for the copy operation. The caller is responsible for aligning the SrcOffset if required.

Count  The number of memory operations to perform. Bytes moved is Width size * Count, starting at DestOffset and SrcOffset.

Description
The CopyMem() function enables a PCI driver to copy one region of PCI memory space to another region of PCI memory space on a PCI controller. This is especially useful for video scroll operations on a memory mapped video buffer.

The memory operations are carried out exactly as requested. The caller is responsible for satisfying any alignment and memory width restrictions that a PCI controller on a platform might require. For example on some platforms, width requests of EfiPciIoWidthUint64 do not work.

If Width is EfiPciIoWidthUint8, EfiPciIoWidthUint16, EfiPciIoWidthUint32, or EfiPciIoWidthUint64, then Count read/write transactions are performed to move the contents of the SrcOffset buffer to the DestOffset buffer. The implementation must be reentrant, and it must handle overlapping SrcOffset and DestOffset buffers. This means that the implementation of CopyMem() must choose the correct direction of the copy operation based on the type of overlap that exists between the SrcOffset and DestOffset buffers. If either the SrcOffset buffer or the DestOffset buffer crosses the top of the processor’s address space, then the result of the copy operation is unpredictable.

The contents of the DestOffset buffer on exit from this service must match the contents of the SrcOffset buffer on entry to this service. Due to potential overlaps, the contents of the SrcOffset buffer may be modified by this service. The following rules can be used to guarantee the correct behavior:
If $DestOffset > SrcOffset$ and $DestOffset < (SrcOffset + Width \text{ size} \times Count)$, then the data should be copied from the $SrcOffset$ buffer to the $DestOffset$ buffer starting from the end of buffers and working toward the beginning of the buffers.

Otherwise, the data should be copied from the $SrcOffset$ buffer to the $DestOffset$ buffer starting from the beginning of the buffers and working toward the end of the buffers.

All the PCI transactions generated by this function are guaranteed to be completed before this function returns. All the PCI write transactions generated by this function will follow the write ordering and completion rules defined in the PCI Specification. However, if the memory-mapped I/O region being accessed by this function has the `EFI_PCI_IO_ATTRIBUTE_MEMORY_CACHED` attribute set, then the transactions will follow the ordering rules defined by the processor architecture.

### Status Codes Returned

<table>
<thead>
<tr>
<th>Code</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>EFI_SUCCESS</td>
<td>The data was copied from one memory region to another memory region.</td>
</tr>
<tr>
<td>EFI_INVALID_PARAMETER</td>
<td>$Width$ is invalid.</td>
</tr>
<tr>
<td>EFI_UNSUPPORTED</td>
<td>$DestBarIndex$ not valid for this PCI controller.</td>
</tr>
<tr>
<td>EFI_UNSUPPORTED</td>
<td>$SrcBarIndex$ not valid for this PCI controller.</td>
</tr>
<tr>
<td>EFI_UNSUPPORTED</td>
<td>The address range specified by $DestOffset$, $Width$, and $Count$ is not</td>
</tr>
<tr>
<td></td>
<td>valid for the PCI BAR specified by $DestBarIndex$.</td>
</tr>
<tr>
<td>EFI_UNSUPPORTED</td>
<td>The address range specified by $SrcOffset$, $Width$, and $Count$ is not</td>
</tr>
<tr>
<td></td>
<td>valid for the PCI BAR specified by $SrcBarIndex$.</td>
</tr>
<tr>
<td>EFI_OUT_OF_RESOURCES</td>
<td>The request could not be completed due to a lack of resources.</td>
</tr>
</tbody>
</table>

### EFI_PCI_IO_PROTOCOL.Map()

**Summary**

Provides the PCI controller–specific addresses needed to access system memory.

**Prototype**

```c
typedef EFI_STATUS
  (EFIAPI *EFI_PCI_IO_PROTOCOL_MAP) ( 
  IN EFI_PCI_IO_PROTOCOL *This, 
  IN EFI_PCI_IO_PROTOCOL_OPERATION Operation, 
  IN VOID *HostAddress, 
  IN OUT UINTN *NumberOfBytes, 
  OUT EFI_PHYSICAL_ADDRESS *DeviceAddress, 
  OUT VOID **Mapping 
); 
```

**Parameters**

- **This**
  A pointer to the EFI_PCI_IO_PROTOCOL instance. Type `EFI_PCI_IO_PROTOCOL` is defined in Section 14.4.

- **Operation**
  Indicates if the bus master is going to read or write to system memory. Type `EFI_PCI_IO_PROTOCOL_OPERATION` is defined in Section 14.4.
**HostAddress**
The system memory address to map to the PCI controller.

**NumberOfBytes**
On input the number of bytes to map. On output the number of bytes that were mapped.

**DeviceAddress**
The resulting map address for the bus master PCI controller to use to access the hosts *HostAddress*. Type `EFI_PHYSICAL_ADDRESS` is defined in `EFI_BOOT_SERVICES.AllocatePool()`. This address cannot be used by the processor to access the contents of the buffer specified by *HostAddress*.

**Mapping**
A resulting value to pass to `Unmap()`.

**Description**
The `Map()` function provides the PCI controller–specific addresses needed to access system memory. This function is used to map system memory for PCI bus master DMA accesses.

All PCI bus master accesses must be performed through their mapped addresses and such mappings must be freed with `Unmap()` when complete. If the bus master access is a single read or write data transfer, then `EfiPciIoOperationBusMasterRead` or `EfiPciIoOperation-BusMasterWrite` is used and the range is unmapped to complete the operation. If performing an `EfiPciIoOperationBusMasterRead` operation, all the data must be present in system memory before the `Map()` is performed. Similarly, if performing an `EfiPciIoOperation-BusMasterWrite`, the data cannot be properly accessed in system memory until `Unmap()` is performed.

Bus master operations that require both read and write access or require multiple host device interactions within the same mapped region must use `EfiPciIoOperation-BusMasterCommonBuffer`. However, only memory allocated via the `AllocateBuffer()` interface can be mapped for this operation type.

In all mapping requests the resulting `NumberOfBytes` actually mapped may be less than the requested amount. In this case, the DMA operation will have to be broken up into smaller chunks. The `Map()` function will map as much of the DMA operation as it can at one time. The caller may have to loop on `Map()` and `Unmap()` in order to complete a large DMA transfer.
Status Codes Returned

<table>
<thead>
<tr>
<th>Status Code</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>EFI_SUCCESS</td>
<td>The range was mapped for the returned NumberOfBytes.</td>
</tr>
<tr>
<td>EFI_INVALID_PARAMETER</td>
<td>Operation is invalid.</td>
</tr>
<tr>
<td>EFI_INVALID_PARAMETER</td>
<td>HostAddress is NULL.</td>
</tr>
<tr>
<td>EFI_INVALID_PARAMETER</td>
<td>NumberOfBytes is NULL.</td>
</tr>
<tr>
<td>EFI_INVALID_PARAMETER</td>
<td>DeviceAddress is NULL.</td>
</tr>
<tr>
<td>EFI_INVALID_PARAMETER</td>
<td>Mapping is NULL.</td>
</tr>
<tr>
<td>EFI_UNSUPPORTED</td>
<td>The HostAddress cannot be mapped as a common buffer.</td>
</tr>
<tr>
<td>EFI_DEVICE_ERROR</td>
<td>The system hardware could not map the requested address.</td>
</tr>
<tr>
<td>EFI_OUT_OF_RESOURCES</td>
<td>The request could not be completed due to a lack of resources.</td>
</tr>
</tbody>
</table>

### EFI_PCI_IO_PROTOCOL.Unmap()

**Summary**
Completes the Map() operation and releases any corresponding resources.

**Prototype**
```c
typedef EFI_STATUS (EFIAPI *EFI_PCI_IO_PROTOCOL_UNMAP) (IN EFI_PCI_IO_PROTOCOL *This, IN VOID *Mapping);
```

**Parameters**
- **This**
  A pointer to the EFI_PCI_IO_PROTOCOL instance. Type EFI_PCI_IO_PROTOCOL is defined in Section 14.4.
- **Mapping**
  The mapping value returned from Map().

**Description**
The Unmap() function completes the Map() operation and releases any corresponding resources. If the operation was an EfiPciIoOperationBusMasterWrite, the data is committed to the target system memory. Any resources used for the mapping are freed.

**Status Codes Returned**

<table>
<thead>
<tr>
<th>Status Code</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>EFI_SUCCESS</td>
<td>The range was unmapped.</td>
</tr>
<tr>
<td>EFI_DEVICE_ERROR</td>
<td>The data was not committed to the target system memory.</td>
</tr>
</tbody>
</table>

### EFI_PCI_IO_PROTOCOL.AllocateBuffer()

**Summary**
Allocates pages that are suitable for an EfiPciIoOperationBusMasterCommonBuffer mapping.
Prototype

```c
typedef EFI_STATUS
    (EFIAPI *EFI_PCI_IO_PROTOCOL_ALLOCATE_BUFFER) (
    IN   EFI_PCI_IO_PROTOCOL   *This,
    IN   EFI_ALLOCATE_TYPE     Type,
    IN   EFI_MEMORY_TYPE       MemoryType,
    IN   UINTN                 Pages,
    OUT  VOID                  **HostAddress,
    IN   UINT64                Attributes
    );
```

Parameters

- **This**: A pointer to the EFI_PCI_IO_PROTOCOL instance. Type EFI_PCI_IO_PROTOCOL is defined in Section 14.4.
- **Type**: This parameter is not used and must be ignored.
- **MemoryType**: The type of memory to allocate, EfiBootServicesData or EfiRuntimeServicesData. Type EFI_MEMORY_TYPE is defined in EFI_BOOT_SERVICES.AllocatePages().
- **Pages**: The number of pages to allocate.
- **HostAddress**: A pointer to store the base system memory address of the allocated range.
- **Attributes**: The requested bit mask of attributes for the allocated range. Only the attributes EFI_PCI_IO_ATTRIBUTE_MEMORY_WRITE_COMBINE, and EFI_PCI_IO_ATTRIBUTE_MEMORY_CACHED may be used with this function. If any other bits are set, then EFI_UNSUPPORTED is returned. This function may choose to ignore this bit mask. The EFI_PCI_IO_ATTRIBUTE_MEMORY_WRITE_COMBINE, and EFI_PCI_IO_ATTRIBUTE_MEMORY_CACHED attributes provide a hint to the implementation that may improve the performance of the calling driver. The implementation may choose any default for the memory attributes including write combining, cached, both, or neither as long as the allocated buffer can be seen equally by both the processor and the PCI bus master.

Description

The AllocateBuffer() function allocates pages that are suitable for an EfiPciIoOperationBusMasterCommonBuffer mapping. This means that the buffer allocated by this function must support simultaneous access by both the processor and a PCI Bus Master. The device address that the PCI Bus Master uses to access the buffer can be retrieved with a call to Map().

If the current attributes of the PCI controller has the EFI_PCI_IO_ATTRIBUTE_DUAL_ADDRESS_CYCLE bit set, then when the buffer allocated by this function is mapped with a call to Map(), the device address that is returned by Map() must be within the 64-bit device address space of the PCI Bus Master. The attributes for a PCI controller can be managed by calling Attributes().
If the current attributes for the PCI controller has the EFI_PCI_IO_ATTRIBUTE_DUAL_ADDRESS_CYCLE bit clear, then when the buffer allocated by this function is mapped with a call to Map(), the device address that is returned by Map() must be within the 32-bit device address space of the PCI Bus Master. The attributes for a PCI controller can be managed by calling Attributes().

If the memory allocation specified by MemoryType and Pages cannot be satisfied, then EFI_OUT_OF_RESOURCES is returned.

**Status Codes Returned**

<table>
<thead>
<tr>
<th>Status Code</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>EFI_SUCCESS</td>
<td>The requested memory pages were allocated.</td>
</tr>
<tr>
<td>EFI_INVALID_PARAMETER</td>
<td>MemoryType is invalid.</td>
</tr>
<tr>
<td>EFI_INVALID_PARAMETER</td>
<td>HostAddress is NULL.</td>
</tr>
<tr>
<td>EFI_UNSUPPORTED</td>
<td>Attributes is unsupported. The only legal attribute bits are MEMORY_WRITE_COMBINE and MEMORY_CACHED.</td>
</tr>
<tr>
<td>EFI_OUT_OF_RESOURCES</td>
<td>The memory pages could not be allocated.</td>
</tr>
</tbody>
</table>

**EFI_PCI_IO_PROTOCOL.FreeBuffer()**

**Summary**
Frees memory that was allocated with AllocateBuffer().

**Prototype**

```c
typedef EFI_STATUS
    (EFIAPI *EFI_PCI_IO_PROTOCOL_FREE_BUFFER) (IN EFI_PCI_IO_PROTOCOL *This,
    IN UINTN Pages,
    IN VOID *HostAddress);
```

**Parameters**

- **This**
  A pointer to the EFI_PCI_IO_PROTOCOL instance. Type EFI_PCI_IO_PROTOCOL is defined in Section 14.4.
- **Pages**
  The number of pages to free.
- **HostAddress**
  The base system memory address of the allocated range.

**Description**
The FreeBuffer() function frees memory that was allocated with AllocateBuffer().
Status Codes Returned

<table>
<thead>
<tr>
<th>Status Code</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>EFI_SUCCESS</td>
<td>The requested memory pages were freed.</td>
</tr>
<tr>
<td>EFI_INVALID_PARAMETER</td>
<td>The memory range specified by HostAddress and Pages was not allocated with AllocateBuffer().</td>
</tr>
</tbody>
</table>

**EFI_PCI_IO_PROTOCOL.Flush()**

**Summary**
Flushes all PCI posted write transactions from a PCI host bridge to system memory.

**Prototype**
```c
typedef EFI_STATUS (EFIAPI *EFI_PCI_IO_PROTOCOL_FLUSH) (IN EFI_PCI_IO_PROTOCOL *This);
```

**Parameters**
- `This` A pointer to the EFI_PCI_IO_PROTOCOL instance. Type EFI_PCI_IO_PROTOCOL is defined in Section 14.4.

**Description**
The `Flush()` function flushes any PCI posted write transactions from a PCI host bridge to system memory. Posted write transactions are generated by PCI bus masters when they perform write transactions to target addresses in system memory.

This function does not flush posted write transactions from any PCI bridges. A PCI controller specific action must be taken to guarantee that the posted write transactions have been flushed from the PCI controller and from all the PCI bridges into the PCI host bridge. This is typically done with a PCI read transaction from the PCI controller prior to calling `Flush()`.

If the PCI controller specific action required to flush the PCI posted write transactions has been performed, and this function returns EFI_SUCCESS, then the PCI bus master’s view and the processor’s view of system memory are guaranteed to be coherent. If the PCI posted write transactions cannot be flushed from the PCI host bridge, then the PCI bus master and processor are not guaranteed to have a coherent view of system memory, and EFI_DEVICE_ERROR is returned.
Status Codes Returned

<table>
<thead>
<tr>
<th>Code</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>EFI_SUCCESS</td>
<td>The PCI posted write transactions were flushed from the PCI host bridge to system memory.</td>
</tr>
<tr>
<td>EFI DEVICE_ERROR</td>
<td>The PCI posted write transactions were not flushed from the PCI host bridge due to a hardware error.</td>
</tr>
</tbody>
</table>

**EFI_PCI_IO_PROTOCOL.GetLocation()**

**Summary**

Retrieves this PCI controller’s current PCI bus number, device number, and function number.

**Prototype**

```c
typedef EFI_STATUS
    (EFIAPI *EFI_PCI_IO_PROTOCOL_GET_LOCATION) ( 
    IN EFI_PCI_IO_PROTOCOL *This, 
    OUT UINTN *SegmentNumber, 
    OUT UINTN *BusNumber, 
    OUT UINTN *DeviceNumber, 
    OUT UINTN *FunctionNumber 
); 
```

**Parameters**

- **This**
  A pointer to the EFI_PCI_IO_PROTOCOL instance. Type `EFI_PCI_IO_PROTOCOL` is defined in Section 14.4.
- **SegmentNumber**
  The PCI controller’s current PCI segment number.
- **BusNumber**
  The PCI controller’s current PCI bus number.
- **DeviceNumber**
  The PCI controller’s current PCI device number.
- **FunctionNumber**
  The PCI controller’s current PCI function number.

**Description**

The `GetLocation()` function retrieves a PCI controller’s current location on a PCI Host Bridge. This is specified by a PCI segment number, PCI bus number, PCI device number, and PCI function number. These values can be used with the PCI Root Bridge I/O Protocol to perform PCI configuration cycles on the PCI controller, or any of its peer PCI controller’s on the same PCI Host Bridge.
Status Codes Returned

<table>
<thead>
<tr>
<th>Code</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>EFI_SUCCESS</td>
<td>The PCI controller location was returned.</td>
</tr>
<tr>
<td>EFI_INVALID_PARAMETER</td>
<td>SegmentNumber is NULL.</td>
</tr>
<tr>
<td>EFI_INVALID_PARAMETER</td>
<td>BusNumber is NULL.</td>
</tr>
<tr>
<td>EFI_INVALID_PARAMETER</td>
<td>DeviceNumber is NULL.</td>
</tr>
<tr>
<td>EFI_INVALID_PARAMETER</td>
<td>FunctionNumber is NULL.</td>
</tr>
</tbody>
</table>

**EFI_PCI_IO_PROTOCOL.Attributes()**

**Summary**
Performs an operation on the attributes that this PCI controller supports. The operations include getting the set of supported attributes, retrieving the current attributes, setting the current attributes, enabling attributes, and disabling attributes.

**Prototype**

```c
typedef EFI_STATUS
    (EFIAPI *EFI_PCI_IO_PROTOCOL_ATTRIBUTES) (  
    IN EFI_PCI_IO_PROTOCOL *This,
    IN EFI_PCI_IO_PROTOCOL_ATTRIBUTE_OPERATION Operation,
    IN UINT64 Attributes,
    OUT UINT64 *Result OPTIONAL
    );
```

**Parameters**

- **This**
  
  A pointer to the EFI_PCI_IO_PROTOCOL instance. Type EFI_PCI_IO_PROTOCOL is defined in Section 14.4.

- **Operation**
  
  The operation to perform on the attributes for this PCI controller. Type EFI_PCI_IO_PROTOCOL_ATTRIBUTE_OPERATION is defined in “Related Definitions” below.

- **Attributes**
  
  The mask of attributes that are used for Set, Enable, and Disable operations. The available attributes are listed in Section 14.4.

- **Result**
  
  A pointer to the result mask of attributes that are returned for the Get and Supported operations. This is an optional parameter that may be NULL for the Set, Enable, and Disable operations. The available attributes are listed in Section 14.4.
Related Definitions

```c
typedef enum {
    EfiPciIoAttributeOperationGet,
    EfiPciIoAttributeOperationSet,
    EfiPciIoAttributeOperationEnable,
    EfiPciIoAttributeOperationDisable,
    EfiPciIoAttributeOperationSupported,
    EfiPciIoAttributeOperationMaximum
} EFI_PCI_IO_PROTOCOL_ATTRIBUTE_OPERATION;
```

**EfiPciIoAttributeOperationGet**

Retrieve the PCI controller’s current attributes, and return them in `Result`. If `Result` is `NULL`, then `EFI_INVALID_PARAMETER` is returned. For this operation, `Attributes` is ignored.

**EfiPciIoAttributeOperationSet**

Set the PCI controller’s current attributes to `Attributes`. If a bit is set in `Attributes` that is not supported by this PCI controller or one of its parent bridges, then `EFI_UNSUPPORTED` is returned. For this operation, `Result` is an optional parameter that may be `NULL`.

**EfiPciIoAttributeOperationEnable**

Enable the attributes specified by the bits that are set in `Attributes` for this PCI controller. Bits in `Attributes` that are clear are ignored. If a bit is set in `Attributes` that is not supported by this PCI controller or one of its parent bridges, then `EFI_UNSUPPORTED` is returned. For this operation, `Result` is an optional parameter that may be `NULL`.

**EfiPciIoAttributeOperationDisable**

Disable the attributes specified by the bits that are set in `Attributes` for this PCI controller. Bits in `Attributes` that are clear are ignored. If a bit is set in `Attributes` that is not supported by this PCI controller or one of its parent bridges, then `EFI_UNSUPPORTED` is returned. For this operation, `Result` is an optional parameter that may be `NULL`.

**EfiPciIoAttributeOperationSupported**

Retrieve the PCI controller’s supported attributes, and return them in `Result`. If `Result` is `NULL`, then `EFI_INVALID_PARAMETER` is returned. For this operation, `Attributes` is ignored.

**Description**

The `Attributes()` function performs an operation on the attributes associated with this PCI controller. If `Operation` is greater than or equal to the maximum operation value, then `EFI_INVALID_PARAMETER` is returned. If `Operation` is `Set` or `Supported`, and `Result` is `NULL`, then `EFI_INVALID_PARAMETER` is returned. If `Operation` is `Set`, `Enable`, or `Disable` for an attribute that
is not supported by the PCI controller, then \texttt{EFI\_UNSUPPORTED} is returned. Otherwise, the operation is performed as described in “Related Definitions” and \texttt{EFI\_SUCCESS} is returned. It is possible for this function to return \texttt{EFI\_UNSUPPORTED} even if the PCI controller supports the attribute. This can occur when the PCI root bridge does not support the attribute. For example, if VGA I/O and VGA Memory transactions cannot be forwarded onto PCI root bridge #2, then a request by a PCI VGA driver to enable the \texttt{VGA\_IO} and \texttt{VGA\_MEMORY} bits will fail even though a PCI VGA controller behind PCI root bridge #2 is able to decode these transactions.

This function will also return \texttt{EFI\_UNSUPPORTED} if more than one PCI controller on the same PCI root bridge has already successfully requested one of the ISA addressing attributes. For example, if one PCI VGA controller had already requested the \texttt{VGA\_IO} and \texttt{VGA\_MEMORY} attributes, then a second PCI VGA controller on the same root bridge cannot succeed in requesting those same attributes. This restriction applies to the ISA-, VGA-, and IDE-related attributes.

\textbf{Status Codes Returned}

<table>
<thead>
<tr>
<th>Status Code</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>\texttt{EFI_SUCCESS}</td>
<td>The operation on the PCI controller’s attributes was completed. If the operation was Get or Supported, then the attribute mask is returned in Result.</td>
</tr>
<tr>
<td>\texttt{EFI_INVALID_PARAMETER}</td>
<td>Operation is greater than or equal to \texttt{EfiPciIoAttributeOperationMaximum}.</td>
</tr>
<tr>
<td>\texttt{EFI_INVALID_PARAMETER}</td>
<td>Operation is Get and Result is NULL.</td>
</tr>
<tr>
<td>\texttt{EFI_INVALID_PARAMETER}</td>
<td>Operation is Supported and Result is NULL.</td>
</tr>
<tr>
<td>\texttt{EFI_UNSUPPORTED}</td>
<td>Operation is Set, and one or more of the bits set in Attributes are not supported by this PCI controller or one of its parent bridges.</td>
</tr>
<tr>
<td>\texttt{EFI_UNSUPPORTED}</td>
<td>Operation is Enable, and one or more of the bits set in Attributes are not supported by this PCI controller or one of its parent bridges.</td>
</tr>
<tr>
<td>\texttt{EFI_UNSUPPORTED}</td>
<td>Operation is Disable, and one or more of the bits set in Attributes are not supported by this PCI controller or one of its parent bridges.</td>
</tr>
</tbody>
</table>

\textbf{EFI\_PCI\_IO\_PROTOCOL.\_GetBarAttributes()}

\textbf{Summary}

Gets the attributes that this PCI controller supports setting on a BAR using \texttt{SetBarAttributes()}, and retrieves the list of resource descriptors for a BAR.

\textbf{Prototype}

\begin{verbatim}
typedef
EFI\_STATUS
(EIFIAPI *\texttt{EFI\_PCI\_IO\_PROTOCOL\_GET\_BAR\_ATTRIBUTES}) (\n    IN EFI\_PCI\_IO\_PROTOCOL \*This, \n    IN UINT8 BarIndex, \n    OUT UINT64 \*Supports OPTIONAL, \n    OUT VOID **Resources OPTIONAL\n);\end{verbatim}
Parameters

This

A pointer to the EFI_PCI_IO_PROTOCOL instance. Type EFI_PCI_IO_PROTOCOL is defined in Section 14.4.

BarIndex

The BAR index of the standard PCI Configuration header to use as the base address for resource range. The legal range for this field is 0..5.

Supports

A pointer to the mask of attributes that this PCI controller supports setting for this BAR with SetBarAttributes(). The list of attributes is listed in Section 14.4. This is an optional parameter that may be NULL.

Resources

A pointer to the resource descriptors that describe the current configuration of this BAR of the PCI controller. This buffer is allocated for the caller with the Boot Service EFI_BOOT_SERVICES.AllocatePool(). It is the caller’s responsibility to free the buffer with the Boot Service EFI_BOOT_SERVICES.FreePool(). See “Related Definitions” below for the resource descriptors that may be used. This is an optional parameter that may be NULL.

Related Definitions

There are only two resource descriptor types from the ACPI Specification that may be used to describe the current resources allocated to BAR of a PCI Controller. These are the QWORD Address Space Descriptor, and the End Tag. The QWORD Address Space Descriptor can describe memory, I/O, and bus number ranges for dynamic or fixed resources. The configuration of a BAR of a PCI Controller is described with one or more QWORD Address Space Descriptors followed by an End Tag. Table 14-10 and Table 14-11 contain these two descriptor types. Please see the ACPI Specification for details on the field values. The ACPI Specification does not define how to use the Address Translation Offset for non-bridge devices. The UEFI Specification is extending the definition of Address Translation Offset to support systems that have different mapping for HostAddress and DeviceAddress. The definition of the Address Space Granularity field in the QWORD Address Space Descriptor differs from the ACPI Specification and the definition in Table 14-10 is the one that must be used.
Table 14-10 QWORD Address Space Descriptor

<table>
<thead>
<tr>
<th>Byte Offset</th>
<th>Byte Length</th>
<th>Data</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x00</td>
<td>0x01</td>
<td>0x8A</td>
<td>QWORD Address Space Descriptor</td>
</tr>
<tr>
<td>0x01</td>
<td>0x02</td>
<td>0x2B</td>
<td>Length of this descriptor in bytes not including the first two fields</td>
</tr>
<tr>
<td>0x03</td>
<td>0x01</td>
<td></td>
<td>Resource Type</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>0 – Memory Range</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>1 – I/O Range</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>2 – Bus Number Range</td>
</tr>
<tr>
<td>0x04</td>
<td>0x01</td>
<td></td>
<td>General Flags</td>
</tr>
<tr>
<td>0x05</td>
<td>0x01</td>
<td></td>
<td>Type Specific Flags</td>
</tr>
<tr>
<td>0x06</td>
<td>0x08</td>
<td></td>
<td>Address Space Granularity. Used to differentiate between a 32-bit memory request and a 64-bit memory request. For a 32-bit memory request, this field should be set to 32. For a 64-bit memory request, this field should be set to 64.</td>
</tr>
<tr>
<td>0x0E</td>
<td>0x08</td>
<td></td>
<td>Address Range Minimum. Starting address of BAR.</td>
</tr>
<tr>
<td>0x16</td>
<td>0x08</td>
<td></td>
<td>Address Range Maximum. Ending address of BAR.</td>
</tr>
<tr>
<td>0x1E</td>
<td>0x08</td>
<td></td>
<td>Address Translation Offset. Offset to apply to the Starting address of a BAR to convert it to a PCI address. This value is zero unless the HostAddress and DeviceAddress for the BAR are different.</td>
</tr>
<tr>
<td>0x26</td>
<td>0x08</td>
<td></td>
<td>Address Length</td>
</tr>
</tbody>
</table>

Table 14-11 End Tag

<table>
<thead>
<tr>
<th>Byte Offset</th>
<th>Byte Length</th>
<th>Data</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x00</td>
<td>0x01</td>
<td>0x79</td>
<td>End Tag</td>
</tr>
<tr>
<td>0x01</td>
<td>0x01</td>
<td>0x00</td>
<td>Checksum. If 0, then checksum is assumed to be valid.</td>
</tr>
</tbody>
</table>

Description

The GetBarAttributes() function returns in Supports the mask of attributes that the PCI controller supports setting for the BAR specified by BarIndex. It also returns in Resources a list of resource descriptors for the BAR specified by BarIndex. Both Supports and Resources are optional parameters. If both Supports and Resources are NULL, then EFI_INVALID_PARAMETER is returned. It is the caller’s responsibility to free Resources with the Boot Service EFI_BOOT_SERVICES.FreePool() when the caller is done with the contents of Resources. If there are not enough resources to allocate Resources, then EFI_OUT_OF_RESOURCES is returned.

If a bit is set in Supports, then the PCI controller supports this attribute type for the BAR specified by BarIndex, and a call can be made to SetBarAttributes() using that attribute type.
Status Codes Returned

<table>
<thead>
<tr>
<th>Code</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>EFI_SUCCESS</td>
<td>If Supports is not NULL, then the attributes that the PCI controller supports are returned in Supports. If Resources is not NULL, then the resource descriptors that the PCI controller is currently using are returned in Resources.</td>
</tr>
<tr>
<td>EFI_OUT_OF_RESOURCES</td>
<td>There are not enough resources available to allocate Resources.</td>
</tr>
<tr>
<td>EFI_UNSUPPORTED</td>
<td>BarIndex not valid for this PCI controller.</td>
</tr>
<tr>
<td>EFI_INVALID_PARAMETER</td>
<td>Both Supports and Attributes are NULL.</td>
</tr>
</tbody>
</table>

EFI_PCI_IO_PROTOCOL.SetBarAttributes()

Summary
Sets the attributes for a range of a BAR on a PCI controller.

Prototype

```c
typedef EFI_STATUS (EFIAPI *EFI_PCI_IO_PROTOCOL_SET_BAR_ATTRIBUTES) (
    IN   EFI_PCI_IO_PROTOCOL   *This,
    IN   UINT64                Attributes,
    IN   UINT8                 BarIndex,
    IN OUT UINT64              *Offset,
    IN OUT UINT64              *Length
);```

Parameters

- **This**
  A pointer to the EFI_PCI_IO_PROTOCOL instance. Type EFI_PCI_IO_PROTOCOL is defined in Section 14.4.

- **Attributes**
  The mask of attributes to set for the resource range specified by BarIndex, Offset, and Length.

- **BarIndex**
  The BAR index of the standard PCI Configuration header to use as the base address for the resource range. The legal range for this field is 0..5.

- **Offset**
  A pointer to the BAR relative base address of the resource range to be modified by the attributes specified by Attributes. On return, *Offset will be set to the actual base address of the resource range. Not all resources can be set to a byte boundary, so the actual base address may differ from the one passed in by the caller.

- **Length**
  A pointer to the length of the resource range to be modified by the attributes specified by Attributes. On return, *Length will be set to the actual length of the resource range. Not all resources can be set to a byte boundary, so the actual length may differ from the one passed in by the caller.
Description

The `SetBarAttributes()` function sets the attributes specified in `Attributes` for the PCI controller on the resource range specified by `BarIndex`, `Offset`, and `Length`. Since the granularity of setting these attributes may vary from resource type to resource type, and from platform to platform, the actual resource range and the one passed in by the caller may differ. As a result, this function may set the attributes specified by `Attributes` on a larger resource range than the caller requested. The actual range is returned in `Offset` and `Length`. The caller is responsible for verifying that the actual range for which the attributes were set is acceptable.

If the attributes are set on the PCI controller, then the actual resource range is returned in `Offset` and `Length`, and `EFI_SUCCESS` is returned. Many of the attribute types also require that the state of the PCI Host Bus Controller and the state of any PCI to PCI bridges between the PCI Host Bus Controller and the PCI Controller to be modified. This function will only return `EFI_SUCCESS` if all of these state changes are made. The PCI Controller may support a combination of attributes, but unless the PCI Host Bus Controller and the PCI to PCI bridges also support that same combination of attributes, then this call will return an error.

If the attributes specified by `Attributes`, or the resource range specified by `BarIndex`, `Offset`, and `Length` are not supported by the PCI controller, then `EFI_UNSUPPORTED` is returned. The set of supported attributes for the PCI controller can be found by calling `GetBarAttributes()`.

If either `Offset` or `Length` is `NULL` then `EFI_INVALID_PARAMETER` is returned.

If there are not enough resources available to set the attributes, then `EFI_OUT_OF_RESOURCES` is returned.

**Status Codes Returned**

<table>
<thead>
<tr>
<th>Status Code</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>EFI_SUCCESS</td>
<td>The set of attributes specified by <code>Attributes</code> for the resource range specified by <code>BarIndex</code>, <code>Offset</code>, and <code>Length</code> were set on the PCI controller, and the actual resource range is returned in <code>Offset</code> and <code>Length</code>.</td>
</tr>
<tr>
<td>EFI_UNSUPPORTED</td>
<td>The set of attributes specified by <code>Attributes</code> is not supported by the PCI controller for the resource range specified by <code>BarIndex</code>, <code>Offset</code>, and <code>Length</code>.</td>
</tr>
<tr>
<td>EFI_INVALID_PARAMETER</td>
<td><code>Offset</code> is <code>NULL</code>.</td>
</tr>
<tr>
<td>EFI_INVALID_PARAMETER</td>
<td><code>Length</code> is <code>NULL</code>.</td>
</tr>
<tr>
<td>EFI_OUT_OF_RESOURCES</td>
<td>There are not enough resources to set the attributes on the resource range specified by <code>BarIndex</code>, <code>Offset</code>, and <code>Length</code>.</td>
</tr>
</tbody>
</table>

14.4.1 PCI Device Paths

An `EFI_PCI_IO_PROTOCOL` must be installed on a handle for its services to be available to PCI device drivers. In addition to the `EFI_PCI_IO_PROTOCOL`, an `EFIDEVICE_PATH_PROTOCOL` must also be installed on the same handle (see chapter 9).

Typically, an ACPI Device Path Node is used to describe a PCI Root Bridge. Depending on the bus hierarchy in the system, additional device path nodes may precede this ACPI Device Path Node. A PCI device path is described with PCI Device Path Nodes. There will be one PCI Device Path node for the PCI...
controller itself, and one PCI Device Path Node for each PCI to PCI Bridge that is between the PCI controller and the PCI Root Bridge.

Table 14-12 shows an example device path for a PCI controller that is located at PCI device number 0x07 and PCI function 0x00, and is directly attached to a PCI root bridge. This device path consists of an ACPI Device Path Node, a PCI Device Path Node, and a Device Path End Structure. The _HID and _UID must match the ACPI table description of the PCI Root Bridge. The shorthand notation for this device path is:

ACPI(PNP0A03,0)/PCI(7,0).

<table>
<thead>
<tr>
<th>Byte Offset</th>
<th>Byte Length</th>
<th>Data</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x00</td>
<td>0x01</td>
<td>0x02</td>
<td>Generic Device Path Header – Type ACPI Device Path</td>
</tr>
<tr>
<td>0x01</td>
<td>0x01</td>
<td>0x01</td>
<td>Sub type – ACPI Device Path</td>
</tr>
<tr>
<td>0x02</td>
<td>0x02</td>
<td>0x0C</td>
<td>Length – 0x0C bytes</td>
</tr>
<tr>
<td>0x04</td>
<td>0x04</td>
<td>0x41D0, 0x0A03</td>
<td>_HID PNP0A03 – 0x41D0 represents the compressed string ‘PNP’ and is encoded in the low order bytes. The compression method is described in the ACPI Specification.</td>
</tr>
<tr>
<td>0x08</td>
<td>0x04</td>
<td>0x0000</td>
<td>_UID</td>
</tr>
<tr>
<td>0x0C</td>
<td>0x01</td>
<td>0x01</td>
<td>Generic Device Path Header – Type Hardware Device Path</td>
</tr>
<tr>
<td>0x0D</td>
<td>0x01</td>
<td>0x01</td>
<td>Sub type – PCI</td>
</tr>
<tr>
<td>0x0E</td>
<td>0x02</td>
<td>0x06</td>
<td>Length – 0x06 bytes</td>
</tr>
<tr>
<td>0x10</td>
<td>0x01</td>
<td>0x00</td>
<td>PCI Function</td>
</tr>
<tr>
<td>0x11</td>
<td>0x01</td>
<td>0x07</td>
<td>PCI Device</td>
</tr>
<tr>
<td>0x12</td>
<td>0x01</td>
<td>0xFF</td>
<td>Generic Device Path Header – Type End of Hardware Device Path</td>
</tr>
<tr>
<td>0x13</td>
<td>0x01</td>
<td>0xFF</td>
<td>Sub type – End of Entire Device Path</td>
</tr>
<tr>
<td>0x14</td>
<td>0x02</td>
<td>0x04</td>
<td>Length – 0x04 bytes</td>
</tr>
</tbody>
</table>

Table 14-13 shows an example device path for a PCI controller that is located behind a PCI to PCI bridge at PCI device number 0x07 and PCI function 0x00. The PCI to PCI bridge is directly attached to a PCI root bridge, and it is at PCI device number 0x05 and PCI function 0x00. This device path consists of an ACPI Device Path Node, two PCI Device Path Nodes, and a Device Path End Structure. The _HID and _UID must match the ACPI table description of the PCI Root Bridge. The shorthand notation for this device path is:

ACPI(PNP0A03,0)/PCI(5,0)/PCI(7,0).
14.4.2 PCI Option ROMs

EFI takes advantage of both the PCI Firmware Specification and the PE/COFF Specification to store EFI images in a PCI Option ROM. There are several rules that must be followed when constructing a PCI Option ROM:

- A PCI Option ROM can be no larger than 16 MiB.
- A PCI Option ROM may contain one or more images.
- Each image must be on a 512-byte boundary.
- Each image must be an even multiple of 512 bytes in length. This means that images that are not an even multiple of 512 bytes in length must be padded to the next 512-byte boundary.
- Legacy Option ROM images begin with a Standard PCI Expansion ROM Header (Table 14-14).
- EFI Option ROM images begin with an EFI PCI Expansion ROM Header (Table 14-18).
- Each image must contain a PCIR data structure in the first 64 KiB of the image.
- The image data for an EFI Option ROM image must begin in the first 64 KiB of the image.
- The image data for an EFI Option ROM image must be a PE/COFF image or a compressed PE/COFF image following the UEFI Compression Algorithm, and referencing Appendix H for the Compression Source Code.

### Table 14-13 PCI Device 7, Function 0 behind PCI to PCI bridge

<table>
<thead>
<tr>
<th>Byte Offset</th>
<th>Byte Length</th>
<th>Data</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x00</td>
<td>0x01</td>
<td>0x02</td>
<td>Generic Device Path Header – Type ACPI Device Path</td>
</tr>
<tr>
<td>0x01</td>
<td>0x01</td>
<td>0x01</td>
<td>Sub type – ACPI Device Path</td>
</tr>
<tr>
<td>0x02</td>
<td>0x02</td>
<td>0x0C</td>
<td>Length – 0x0C bytes</td>
</tr>
<tr>
<td>0x04</td>
<td>0x04</td>
<td>0x41D0, 0x0A03</td>
<td>_HID PNP0A03 – 0x41D0 represents the compressed string ‘PNP’ and is encoded in the low order bytes. The compression method is described in the ACPI Specification.</td>
</tr>
<tr>
<td>0x08</td>
<td>0x04</td>
<td>0x0000</td>
<td>_UID</td>
</tr>
<tr>
<td>0x0C</td>
<td>0x01</td>
<td>0x01</td>
<td>Generic Device Path Header – Type Hardware Device Path</td>
</tr>
<tr>
<td>0x0D</td>
<td>0x01</td>
<td>0x01</td>
<td>Sub type – PCI</td>
</tr>
<tr>
<td>0x0E</td>
<td>0x02</td>
<td>0x06</td>
<td>Length – 0x06 bytes</td>
</tr>
<tr>
<td>0x10</td>
<td>0x01</td>
<td>0x00</td>
<td>PCI Function</td>
</tr>
<tr>
<td>0x11</td>
<td>0x01</td>
<td>0x05</td>
<td>PCI Device</td>
</tr>
<tr>
<td>0x12</td>
<td>0x01</td>
<td>0x01</td>
<td>Generic Device Path Header – Type Hardware Device Path</td>
</tr>
<tr>
<td>0x13</td>
<td>0x01</td>
<td>0x01</td>
<td>Sub type – PCI</td>
</tr>
<tr>
<td>0x14</td>
<td>0x02</td>
<td>0x06</td>
<td>Length – 0x06 bytes</td>
</tr>
<tr>
<td>0x16</td>
<td>0x01</td>
<td>0x00</td>
<td>PCI Function</td>
</tr>
<tr>
<td>0x17</td>
<td>0x01</td>
<td>0x07</td>
<td>PCI Device</td>
</tr>
<tr>
<td>0x18</td>
<td>0x01</td>
<td>0xFF</td>
<td>Generic Device Path Header – Type End of Hardware Device Path</td>
</tr>
<tr>
<td>0x19</td>
<td>0x01</td>
<td>0xFF</td>
<td>Sub type – End of Entire Device Path</td>
</tr>
<tr>
<td>0x1A</td>
<td>0x02</td>
<td>0x04</td>
<td>Length – 0x04 bytes</td>
</tr>
</tbody>
</table>
The PCIR data structure must begin on a 4-byte boundary.

If the PCI Option ROM contains a Legacy Option ROM image, it must be the first image.

The images are placed in the PCI Option ROM in order from highest to lowest priority. This priority is used to build the ordered list of Driver Image Handles that are produced by the Bus Specific Driver Override Protocol for a PCI Controller.

When PCI device provides an EFI option ROM that is signed in accordance with Chapter 27, use of UEFI Compression Algorithm storage option is preferred. When performing signature validation upon compressed driver, the size returned by `EFI_DECOMPRESS_PROTOCOL.GetInfo()` will be used as driver size and input to signature validation process. Thus any post-driver padding required to reach exact multiple of 512 bytes per Figure 14-15 is ignored by signature validation.

When PCI device provides an EFI option ROM that is signed in accordance with Chapter 27 and stored uncompressed, the end of the driver for signature validation will be the assumed to be the 512-byte boundary indicated by the ‘Initialization Size’ value in the EFI PCI Expansion ROM Header (see Table 14-16). As the signed driver may not exactly fill the indicated ‘Initialization Size’, it is recommended that the value ‘Offset to EFI Image’ (also Table 14-16) be adjusted to ensure the last byte of the signed, uncompressed driver, coincides with the end of the ROM as indicated by ‘Initialization Size’. And any required padding bytes are to be inserted ahead of the signed uncompressed driver image.

There are several options available when building a PCI option ROM for a PCI adapter. A PCI Option ROM can choose to support only a legacy PC-AT platform, only an EFI compliant platform, or both. This flexibility allows a migration path from adapters that support only legacy PC-AT platforms, to adapters that support both PC-AT platforms and EFI compliant platforms, to adapters that support only EFI compliant platforms. The following is a list of the image combinations that may be placed in a PCI option ROM. This is not an exhaustive list. Instead, it provides what will likely be the most common PCI option ROM layouts. EFI compliant system firmware must work with all of these PCI option ROM layouts, plus any other layouts that are possible within the PCI Firmware Specification. The format of a Legacy Option ROM image is defined in the PCI Firmware Specification.

- Legacy Option ROM image
- IA-32 UEFI driver
- x64 UEFI driver
- AArch32 UEFI driver
- AArch64 UEFI driver
- RISCV32 UEFI driver
- RISCV64 UEFI driver
- RISCV128 UEFI driver
- Legacy Option ROM image + x64 UEFI driver
- Legacy Option ROM image + x64 UEFI driver + AArch64 UEFI driver
- x64 UEFI driver + AArch64 UEFI driver
- Itanium Processor Family UEFI driver
- EBC Driver
In addition to combinations of UEFI drivers with different processor binding, it is also possible to include multiple drivers of different function but the same processor binding. When processing option ROM contents, all drivers of appropriate processor binding type must be loaded and added to ordered list of drivers previously mentioned.

It is also possible to place a application written to this specification in a PCI Option ROM. However, the PCI Bus Driver will ignore these images. The exact mechanism by which applications can be loaded and executed from a PCI Option ROM is outside the scope of this document.

### Table 14-14 Standard PCI Expansion ROM Header (Example from PCI Firmware Specification 3.0)

<table>
<thead>
<tr>
<th>Offset</th>
<th>Byte Length</th>
<th>Value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x00</td>
<td>1</td>
<td>0x55</td>
<td>ROM Signature, byte 1</td>
</tr>
<tr>
<td>0x01</td>
<td>1</td>
<td>0xAA</td>
<td>ROM Signature, byte 2</td>
</tr>
<tr>
<td>0x02-0x17</td>
<td>22</td>
<td>XX</td>
<td>Reserved per processor architecture unique data</td>
</tr>
<tr>
<td>0x18-0x19</td>
<td>2</td>
<td>XX</td>
<td>Pointer to PCIR Data Structure</td>
</tr>
</tbody>
</table>

### Table 14-15 PCI Expansion ROM Code Types (Example from PCI Firmware Specification 3.0)

<table>
<thead>
<tr>
<th>Code Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x00</td>
<td>IA-32, PC-AT compatible</td>
</tr>
<tr>
<td>0x01</td>
<td>Open Firmware standard for PCI</td>
</tr>
<tr>
<td>0x02</td>
<td>Hewlett-Packard PA RISC</td>
</tr>
<tr>
<td>0x03</td>
<td>EFI Image</td>
</tr>
<tr>
<td>0x04-0xFF</td>
<td>Reserved</td>
</tr>
</tbody>
</table>

### Table 14-16 EFI PCI Expansion ROM Header

<table>
<thead>
<tr>
<th>Offset</th>
<th>Byte Length</th>
<th>Value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x00</td>
<td>1</td>
<td>0x55</td>
<td>ROM Signature, byte 1</td>
</tr>
<tr>
<td>0x01</td>
<td>1</td>
<td>0xAA</td>
<td>ROM Signature, byte 2</td>
</tr>
<tr>
<td>0x02</td>
<td>2</td>
<td>XXXX</td>
<td>Initialization Size – size of this image in units of 512 bytes. The size includes this header.</td>
</tr>
<tr>
<td>0x04</td>
<td>4</td>
<td>0x0EF1</td>
<td>Signature from EFI image header</td>
</tr>
<tr>
<td>0x08</td>
<td>2</td>
<td>XX</td>
<td>Subsystem value for EFI image header</td>
</tr>
<tr>
<td>0x0a</td>
<td>2</td>
<td>XX</td>
<td>Machine type from EFI image header</td>
</tr>
<tr>
<td>0x0c</td>
<td>2</td>
<td>XX</td>
<td>Compression type</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>0x0000 - The image is uncompressed</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>0x0001 - The image is compressed. See the UEFI Compression Algorithm and Appendix H.</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>0x0002 - 0xFFFF - Reserved</td>
</tr>
<tr>
<td>0x0e</td>
<td>8</td>
<td>0x00</td>
<td>Reserved</td>
</tr>
<tr>
<td>0x16</td>
<td>2</td>
<td>XX</td>
<td>Offset to EFI Image</td>
</tr>
<tr>
<td>0x18</td>
<td>2</td>
<td>XX</td>
<td>Offset to PCIR Data Structure</td>
</tr>
</tbody>
</table>
14.4.2.1 PCI Bus Driver Responsibilities

A PCI Bus Driver must scan a PCI Option ROM for PCI Device Drivers. If a PCI Option ROM is found during PCI Enumeration, then a copy of the PCI Option ROM is placed in a memory buffer. The PCI Bus Driver will use the memory copy of the PCI Option ROM to search for UEFI drivers after PCI Enumeration. The PCI Bus Driver will search the list of images in a PCI Option ROM for the ones that have a Code Type of 0x03 in the PCIR Data Structure, and a Signature of 0xEF1 in the EFI PCI Expansion ROM Header. Then, it will examine the Subsystem Type of the EFI PCI Expansion ROM Header. If the Subsystem Type is IMAGE_SUBSYSTEM_EFI_BOOT_SERVICE_DRIVER (11) or IMAGE_SUBSYSTEM_EFI_RUNTIME_DRIVER (12), then the PCI Bus Driver can load the PCI Device Driver from the PCI Option ROM. The Offset to EFI Image Header field of the EFI PCI Expansion ROM Header is used to get a pointer to the beginning of the PE/COFF image in the PCI Option ROM. The PE/COFF image may have been compressed using the UEFI Compression Algorithm. If it has been compressed, then the PCI Bus Driver must decompress the driver to a memory buffer. The Boot Service EFI_BOOT_SERVICES.LoadImage() can then be used to load the driver. All UEFI driver images discovered in the PCI Option ROM and meeting these requirements must be processed and loaded via LoadImage(). If the platform does not support the Machine Type of the driver, then LoadImage() may fail.

It is the PCI Bus Driver’s responsibility to verify that the Expansion ROM Header and PCIR Data Structure are valid. It is the responsibly of the Boot Service LoadImage() to verify that the PE/COFF image is valid. The Boot Service LoadImage() may fail for several reasons including a corrupt PE/COFF image or an unsupported Machine Type.

If a PCI Option ROM contains one or more UEFI images, then the PCI Bus Driver must install an instance of the EFI_LOAD_FILE2_PROTOCOL on the PCI controller handle. Then, when the PCI Bus Driver loads a PE/COFF image from a PCI Option ROM using the Boot Service LoadImage(), the PCI Bus Driver must provide the device path of the image being loaded. The device path of an image loaded from a PCI Option ROM must be the device path to the PCI Controller to which the PCI Option ROM is attached followed by a Relative Offset Range node. The Starting Offset field of the Relative Offset Range node must be the byte offset from the beginning of the PCI Option ROM to the beginning of the EFI Option ROM image, and the Ending Offset field of the Relative Offset Range node must be the byte offset from the beginning of the PCI Option ROM to the end of the EFI Option ROM image. The table below shows an example device path for an EFI driver loaded from a PCI Option ROM. The EFI Driver starts at offset 0x8000 into the PCI Option ROM and is 0x2000 bytes long. The shorthand notation for this device path is:

\[ \text{PciRoot(0)/PCI(5,0)/PCI(7,0)/ Offset(0x8000,0xFFFF)} \]
Table 14-17 Device Path for an EFI Driver loaded from PCI Option ROM

<table>
<thead>
<tr>
<th>Byte Offset</th>
<th>Byte Length</th>
<th>Data</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x00</td>
<td>0x01</td>
<td>0x02</td>
<td><strong>Generic Device Path Header</strong> – Type ACPI Device Path</td>
</tr>
<tr>
<td>0x01</td>
<td>0x01</td>
<td>0x01</td>
<td>Sub type – ACPI Device Path</td>
</tr>
<tr>
<td>0x02</td>
<td>0x02</td>
<td>0x0C</td>
<td>Length – 0x0C bytes</td>
</tr>
<tr>
<td>0x04</td>
<td>0x04</td>
<td>0x41D0, 0x0A03</td>
<td>_HID PNP0A03 – 0x41D0 represents the compressed string ‘PNP’ and is encoded in the low order bytes. The compression method is described in the ACPI Specification.</td>
</tr>
<tr>
<td>0x08</td>
<td>0x04</td>
<td>0x0000</td>
<td>_UID</td>
</tr>
<tr>
<td>0x0C</td>
<td>0x01</td>
<td>0x01</td>
<td><strong>Generic Device Path Header</strong> – Type Hardware Device Path</td>
</tr>
<tr>
<td>0x0D</td>
<td>0x01</td>
<td>0x01</td>
<td>Sub type – PCI</td>
</tr>
<tr>
<td>0x0E</td>
<td>0x02</td>
<td>0x06</td>
<td>Length – 0x06 bytes</td>
</tr>
<tr>
<td>0x10</td>
<td>0x01</td>
<td>0x00</td>
<td>PCI Function</td>
</tr>
<tr>
<td>0x11</td>
<td>0x01</td>
<td>0x05</td>
<td>PCI Device</td>
</tr>
<tr>
<td>0x12</td>
<td>0x01</td>
<td>0x01</td>
<td><strong>Generic Device Path Header</strong> – Type Hardware Device Path</td>
</tr>
<tr>
<td>0x13</td>
<td>0x01</td>
<td>0x01</td>
<td>Sub type – PCI</td>
</tr>
<tr>
<td>0x14</td>
<td>0x02</td>
<td>0x06</td>
<td>Length – 0x06 bytes</td>
</tr>
<tr>
<td>0x16</td>
<td>0x01</td>
<td>0x00</td>
<td>PCI Function</td>
</tr>
<tr>
<td>0x17</td>
<td>0x01</td>
<td>0x07</td>
<td>PCI Device</td>
</tr>
<tr>
<td>0x18</td>
<td>0x01</td>
<td>0x04</td>
<td><strong>Generic Device Path Header</strong> – Type Media Device Path</td>
</tr>
<tr>
<td>0x19</td>
<td>0x01</td>
<td>0x08</td>
<td>Sub type – Relative Offset Range</td>
</tr>
<tr>
<td>0x1A</td>
<td>0x02</td>
<td>0x14</td>
<td>Length – 0x14 bytes</td>
</tr>
<tr>
<td>0x1C</td>
<td>0x08</td>
<td>0x8000</td>
<td>Start Address – Offset into PCI Option ROM</td>
</tr>
<tr>
<td>0x24</td>
<td>0x08</td>
<td>0x9FFF</td>
<td>End Address – Offset into PCI Option ROM</td>
</tr>
<tr>
<td>0x2C</td>
<td>0x01</td>
<td>0xFF</td>
<td><strong>Generic Device Path Header</strong> – Type End of Hardware Device Path</td>
</tr>
<tr>
<td>0x2D</td>
<td>0x01</td>
<td>0xFF</td>
<td>Sub type – End of Entire Device Path</td>
</tr>
<tr>
<td>0x2E</td>
<td>0x02</td>
<td>0x04</td>
<td>Length – 0x04 bytes</td>
</tr>
</tbody>
</table>

The PCI Option ROM search may produce one or more Driver Image Handles for the PCI Controller that is associated with the PCI Option ROM. The PCI Bus Driver is responsible for producing a Bus Specific Driver Override Protocol instance for every PCI Controller has a PCI Option ROM that contains one or more UEFI Drivers. The Bus Specific Driver Override Protocol produces an ordered list of Driver Image Handles. The order that the UEFI Drivers are placed in the PCI Option ROM is the order of Driver Image Handles that must be returned by the Bus Specific Driver Override Protocol. This gives the party that builds the PCI Option ROM control over the order that the drivers are used in the Boot Service `EFI_BOOT_SERVICES.ConnectController()`.

### 14.4.2.2 PCI Device Driver Responsibilities

A PCI Device Driver should not be designed to care where it is stored. It can reside in a PCI Option ROM, the system's motherboard ROM, a hard drive, a CD-ROM drive, etc. All PCI Device Drivers are compiled and linked to generate a PE/COFF image. When a PE/COFF image is placed in a PCI Option ROM, it must
follow the rules outlined in Section 14.4.2. The recommended image layout is to insert an EFI PCI Expansion ROM Header and a PCIR Data Structure in front of the PE/COFF image, and pad the entire image up to the next 512-byte boundary. Figure 14-15 shows the format of a single PCI Device Driver that can be added to a PCI Option ROM.

Following are recommended layouts and flow charts for various types of driver signage and compression states for PCI device driver images. Figure 14-15 shows an unsigned layout.

---

**Figure 14-15 Unsigned PCI Driver Image Layout**

---

PCI Device Driver Image (Unsigned)

- EFI PCI Expansion ROM Header
- Two (2) Bytes of Padding
- PCIR Data Structure
- PE/COFF Image of PCI Device Driver
- Padding to next 512-byte boundary
Figure 14-16 and Figure 14-17 show a signed and compressed PCI device driver image flow chart and layout, respectively.
Figure 14-17 Signed and Compressed PCI Driver Image Layout
Figure 14-18 and Figure 14-19 show a signed but not compressed flow chart and a signed and uncompressed PCI device driver image layout, respectively.

Figure 14-18 Signed but not Compressed PCI Driver Image Flow
Figure 14-19 Signed and Uncompressed PCI Driver Image Layout

The field values for the EFI PCI Expansion ROM Header and the PCIR Data Structure would be as follows in this recommended PCI Driver image layout. An image must start at a 512-byte boundary, and the end of the image must be padded to the next 512-byte boundary.
<table>
<thead>
<tr>
<th>Offset</th>
<th>Byte Length</th>
<th>Value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x00</td>
<td>1</td>
<td>0x55</td>
<td>ROM Signature, byte 1</td>
</tr>
<tr>
<td>0x01</td>
<td>1</td>
<td>0xAA</td>
<td>ROM Signature, byte 2</td>
</tr>
<tr>
<td>0x02</td>
<td>2</td>
<td>XXXX</td>
<td>Initialization Size — size of this image in units of 512 bytes. The size includes this header</td>
</tr>
<tr>
<td>0x04</td>
<td>4</td>
<td>0x0EF1</td>
<td>Signature from EFI image header</td>
</tr>
<tr>
<td>0x08</td>
<td>2</td>
<td>XX</td>
<td>Subsystem Value from the PCI Driver’s PE/COFF Image Header</td>
</tr>
<tr>
<td></td>
<td></td>
<td>0x0B</td>
<td>Subsystem Value for an EFI Boot Service Driver</td>
</tr>
<tr>
<td></td>
<td></td>
<td>0x0C</td>
<td>Subsystem Value for an EFI Runtime Driver</td>
</tr>
<tr>
<td>0x0a</td>
<td>2</td>
<td>XX</td>
<td>Machine type from the PCI Driver’s PE/COFF Image Header</td>
</tr>
<tr>
<td></td>
<td></td>
<td>0x014C</td>
<td>IA-32 Machine Type</td>
</tr>
<tr>
<td></td>
<td></td>
<td>0x0200</td>
<td>Itanium processor type</td>
</tr>
<tr>
<td></td>
<td></td>
<td>0x0EBC</td>
<td>EFI Byte Code (EBC) Machine Type</td>
</tr>
<tr>
<td></td>
<td></td>
<td>0x8664</td>
<td>X64 Machine Type</td>
</tr>
<tr>
<td></td>
<td></td>
<td>0x01C2</td>
<td>ARM Machine Type</td>
</tr>
<tr>
<td></td>
<td></td>
<td>0xAA64</td>
<td>ARM 64-bit Machine Type</td>
</tr>
<tr>
<td>0x0C</td>
<td>2</td>
<td>XXXX</td>
<td>Compression Type</td>
</tr>
<tr>
<td></td>
<td></td>
<td>0x0000</td>
<td>Uncompressed</td>
</tr>
<tr>
<td></td>
<td></td>
<td>0x0001</td>
<td>Compressed following the UEFI Compression Algorithm.</td>
</tr>
<tr>
<td>0x0E</td>
<td>8</td>
<td>0x00</td>
<td>Reserved</td>
</tr>
<tr>
<td>0x16</td>
<td>2</td>
<td>0x0034</td>
<td>Offset to EFI Image</td>
</tr>
<tr>
<td>0x18</td>
<td>2</td>
<td>0x001C</td>
<td>Offset to PCIR Data Structure</td>
</tr>
<tr>
<td>0x1A</td>
<td>2</td>
<td>0x0000</td>
<td>Padding to align PCIR Data Structure on a 4 byte boundary</td>
</tr>
<tr>
<td>0x1C</td>
<td>4</td>
<td>'PCIR'</td>
<td>PCIR Data Structure Signature</td>
</tr>
<tr>
<td>0x20</td>
<td>2</td>
<td>XXXX</td>
<td>Vendor ID from the PCI Controller’s Configuration Header</td>
</tr>
<tr>
<td>0x22</td>
<td>2</td>
<td>XXXX</td>
<td>Device ID from the PCI Controller’s Configuration Header</td>
</tr>
<tr>
<td>0x24</td>
<td>2</td>
<td>0x0000</td>
<td>Reserved</td>
</tr>
<tr>
<td>0x26</td>
<td>2</td>
<td>0x0018</td>
<td>The length if the PCIR Data Structure in bytes</td>
</tr>
<tr>
<td>0x28</td>
<td>1</td>
<td>0x00</td>
<td>PCIR Data Structure Revision. Value for PCI 2.2 Option ROM</td>
</tr>
<tr>
<td>0x29</td>
<td>3</td>
<td>XXXX</td>
<td>Class Code from the PCI Controller’s Configuration Header</td>
</tr>
<tr>
<td>0x2C</td>
<td>2</td>
<td>XXXX</td>
<td>Code Image Length in units of 512 bytes. Same as Initialization Size</td>
</tr>
<tr>
<td>0x2E</td>
<td>2</td>
<td>XXXX</td>
<td>Revision Level of the Code/Data. This field is ignored</td>
</tr>
<tr>
<td>0x30</td>
<td>1</td>
<td>0x03</td>
<td>Code Type</td>
</tr>
<tr>
<td>0x31</td>
<td>1</td>
<td>XX</td>
<td>Indicator. Bit 7 clear means another image follows. Bit 7 set means that this image is the last image in the PCI Option ROM. Bits 0–6 are reserved.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>0x00</td>
<td>Additional images follow this image in the PCI Option ROM</td>
</tr>
<tr>
<td></td>
<td></td>
<td>0x80</td>
<td>This image is the last image in the PCI Option ROM</td>
</tr>
<tr>
<td>0x32</td>
<td>2</td>
<td>0x0000</td>
<td>Reserved</td>
</tr>
<tr>
<td>0x34</td>
<td>X</td>
<td>XXXX</td>
<td>The beginning of the PCI Device Driver’s PE/COFF Image</td>
</tr>
</tbody>
</table>
14.4.3 Nonvolatile Storage

A PCI adapter may contain some form of nonvolatile storage. Since there are no standard access mechanisms for nonvolatile storage on PCI adapters, the PCI I/O Protocol does not provide any services for nonvolatile storage. However, a PCI Device Driver may choose to implement its own access mechanisms. If there is a private channel between a PCI Controller and a nonvolatile storage device, a PCI Device Driver can use it for configuration options or vital product data.

Note: The fields RomImage and RomSize in the PCI I/O Protocol do not provide direct access to the PCI Option ROM on a PCI adapter. Instead, they provide access to a copy of the PCI Option ROM in memory. If the contents of the RomImage are modified, only the memory copy is updated. If a vendor wishes to update the contents of a PCI Option ROM, they must provide their own utility or driver to perform this task. There is no guarantee that the BAR for the PCI Option ROM is valid at the time that the utility or driver may execute, so the utility or driver must provide the code required to gain write access to the PCI Option ROM contents. The algorithm for gaining write access to a PCI Option ROM is both platform specific and adapter specific, so it is outside the scope of this document.

14.4.4 PCI Hot-Plug Events

It is possible to design a PCI Bus Driver to work with PCI Bus that conforms to the PCI Hot-Plug Specification. There are two levels of functionality that could be provided in the preboot environment. The first is to initialize the PCI Hot-Plug capable bus so it can be used by an operating system that also conforms to the PCI Hot-Plug Specification. This only affects the PCI Enumeration that is performed in either the PCI Bus Driver’s initialization, or a firmware component that executes prior to the PCI Bus Driver’s initialization. None of the PCI Device Drivers need to be aware of the fact that a PCI Controller may exist in a slot that is capable of a hot-plug event. Also, the addition, removal, and replacement of PCI adapters in the preboot environment would not be allowed.

The second level of functionality is to actually implement the full hot-plug capability in the PCI Bus Driver. This is not recommended because it adds a great deal of complexity to the PCI Bus Driver design with very little added value. However, there is nothing about the PCI Driver Model that would preclude this implementation. It would require using an event based periodic timer to monitor the hot-plug capable slots, and take advantage of the EFI_BOOT_SERVICES.ConnectController()and EFI_BOOT_SERVICES.DisconnectController() Boot Services to dynamically start and stop the drivers that manage the PCI controller that is being added, removed, or replaced. If the EFI_BOOT_SERVICES.DisconnectController() Boot Service fails it must be retried via a periodic timer.
The intent of this chapter is to specify a method of providing direct access to SCSI devices. These protocols provide services that allow a generic driver to produce the Block I/O protocol for SCSI disk devices, and allows an EFI utility to issue commands to any SCSI device. The main reason to provide such an access is to enable S.M.A.R.T. functionality during POST (i.e., issuing Mode Sense, Mode Select, and Log Sense to SCSI devices). This is accomplished by using a generic API such as SCSI Pass Thru. The use of this method will enable additional functionality in the future without modifying the EFI SCSI Pass Thru driver. SCSI Pass Thru is not limited to SCSI channels. It is applicable to all channel technologies that utilize SCSI commands such as SCSI, ATAPI, and Fibre Channel. This chapter describes the SCSI Driver Model. This includes the behavior of SCSI Bus Drivers, the behavior of SCSI Device Drivers, and a detailed description of the SCSI I/O Protocol. This chapter provides enough material to implement a SCSI Bus Driver, and the tools required to design and implement SCSI Device Drivers. It does not provide any information on specific SCSI devices.

15.1 SCSI Driver Model Overview

The EFI SCSI Driver Stack includes the SCSI Pass Thru Driver, SCSI Bus Driver and individual SCSI Device Drivers.

SCSI Pass Thru Driver: A SCSI Pass Through Driver manages a SCSI Host Controller that contains one or more SCSI Buses. It creates SCSI Bus Controller Handles for each SCSI Bus, and attaches Extended SCSI Pass Thru Protocol and Device Path Protocol to each handle the driver produced. Please refer to Section 15.7 and Appendix G for details about the Extended SCSI Pass Thru Protocol.

SCSI Bus Driver: A SCSI Bus Driver manages a SCSI Bus Controller Handle that is created by SCSI Pass Thru Driver. It creates SCSI Device Handles for each SCSI Device Controller detected during SCSI Bus Enumeration, and attaches SCSI I/O Protocol and Device Path Protocol to each handle the driver produced.

SCSI Device Driver: A SCSI Device Driver manages one kind of SCSI Device. Device handles for SCSI Devices are created by SCSI Bus Drivers. A SCSI Device Driver could be a bus driver itself, and may create child handles. But most SCSI Device Drivers will be device drivers that do not create new handles. For the pure device driver, it attaches protocol instance to the device handle of the SCSI Device. These protocol instances are I/O abstractions that allow the SCSI Device to be used in the pre-boot environment. The most common I/O abstractions are used to boot an EFI compliant OS.

15.2 SCSI Bus Drivers

A SCSI Bus Driver manages a SCSI Bus Controller Handle. A SCSI Bus Controller Handle is created by a SCSI Pass Thru Driver and is abstracted in software with the Extended SCSI Pass Thru Protocol. A SCSI Bus Driver will manage handles that contain this protocol. Figure 15-1 shows an example device handle for a SCSI Bus handle. It contains a Device Path Protocol instance and a Extended SCSI Pass Thru Protocol Instance.
15.2.1 Driver Binding Protocol for SCSI Bus Drivers

The Driver Binding Protocol contains three services. These are `Supported()`, `Start()`, and `Stop()`. `Supported()` tests to see if the SCSI Bus Driver can manage a device handle. A SCSI Bus Driver can only manage device handle that contain the Device Path Protocol and the Extended SCSI Pass Thru Protocol, so a SCSI Bus Driver must look for these two protocols on the device handle that is being tested.

The `Start()` function tells the SCSI Bus Driver to start managing a device handle. The device handle should support the protocols shown in Figure 15-1. The Extended SCSI Pass Thru Protocol provides information about a SCSI Channel and the ability to communicate with any SCSI devices attached to that SCSI Channel.

The SCSI Bus Driver has the option of creating all of its children in one call to `Start()`, or spreading it across several calls to `Start()`. In general, if it is possible to design a bus driver to create one child at a time, it should do so to support the rapid boot capability in the UEFI Driver Model. Each of the child device handles created in `Start()` must contain a Device Path Protocol instance, and a SCSI I/O protocol instance. The SCSI I/O Protocol is described in Section 15.4 and Section 14.4. The format of device paths for SCSI Devices is described in Section 15.5. Figure 15-2 shows an example child device handle that is created by a SCSI Bus Driver for a SCSI Device.

A SCSI Bus Driver must perform several steps to manage a SCSI Bus.
1. Scan for the SCSI Devices on the SCSI Channel that connected to the SCSI Bus Controller. If a request is being made to scan only one SCSI Device, then only looks for the one specified. Create a device handle for the SCSI Device found.

2. Install a Device Path Protocol instance and a SCSI I/O Protocol instance on the device handle created for each SCSI Device.

The **Stop()** function tells the SCSI Bus Driver to stop managing a SCSI Bus. The **Stop()** function can destroy one or more of the device handles that were created on a previous call to **Start()**. If all of the child device handles have been destroyed, then **Stop()** will place the SCSI Bus Controller in a quiescent state. The functionality of **Stop()** mirrors **Start()**.

### 15.2.2 SCSI Enumeration

The purpose of the SCSI Enumeration is only to scan for the SCSI Devices attached to the specific SCSI channel. The SCSI Bus driver need not allocate resources for SCSI Devices (like PCI Bus Drivers do), nor need it connect a SCSI Device with its Device Driver (like USB Bus Drivers do). The details of the SCSI Enumeration is implementation specific, thus is out of the scope of this document.

### 15.3 SCSI Device Drivers

SCSI Device Drivers manage SCSI Devices. Device handles for SCSI Devices are created by SCSI Bus Drivers. A SCSI Device Driver could be a bus driver itself, and may create child handles. But most SCSI Device Drivers will be device drivers that do not create new handles. For the pure device driver, it attaches protocol instance to the device handle of the SCSI Device. These protocol instances are I/O abstractions that allow the SCSI Device to be used in the pre-boot environment. The most common I/O abstractions are used to boot an EFI compliant OS.

#### 15.3.1 Driver Binding Protocol for SCSI Device Drivers

The Driver Binding Protocol contains three services. These are **Supported()**, **Start()**, and **Stop()**. **Supported()** tests to see if the SCSI Device Driver can manage a device handle. A SCSI Device Driver can only manage device handle that contain the Device Path Protocol and the SCSI I/O Protocol, so a SCSI Device Driver must look for these two protocols on the device handle that is being tested. In addition, it needs to check to see if the device handle represents a SCSI Device that SCSI Device Driver knows how to manage. This is typically done by using the services of the SCSI I/O Protocol to see whether the device information retrieved is supported by the device driver.

The **Start()** function tells the SCSI Device Driver to start managing a SCSI Device. A SCSI Device Driver could be a bus driver itself, and may create child handles. But most SCSI Device Drivers will be device drivers that do not create new handles. For the pure device driver, it installs one or more addition protocol instances on the device handle for the SCSI Device.

The **Stop()** function mirrors the **Start()** function, so the **Stop()** function completes any outstanding transactions to the SCSI Device and removes the protocol interfaces that were installed in **Start()**.

### 15.4 EFI SCSI I/O Protocol

This section defines the EFI SCSI I/O protocol. This protocol is used by code, typically drivers, running in the EFI boot services environment to access SCSI devices. In particular, functions for managing devices on SCSI buses are defined here.
The interfaces provided in the \texttt{EFI_SCSI_IO_PROTOCOL} are for performing basic operations to access SCSI devices.

**EFI_SCSI_IO_PROTOCOL**

This section provides a detailed description of the \texttt{EFI_SCSI_IO_PROTOCOL}.

**Summary**

Provides services to manage and communicate with SCSI devices.

**GUID**

\begin{verbatim}
#define EFI_SCSI_IO_PROTOCOL_GUID \
{0x932f47e6,0x2362,0x4002,\} \
{0x80,0x3e,0x3c,0xd5,0x4b,0x13,0x8f,0x85}}
\end{verbatim}

**Protocol Interface Structure**

\begin{verbatim}
typedef struct _EFI_SCSI_IO_PROTOCOL {
    EFI_SCSI_IO_PROTOCOL_GET_DEVICE_TYPE GetDeviceType;
    EFI_SCSI_IO_PROTOCOL_GET_DEVICE_LOCATION GetDeviceLocation;
    EFI_SCSI_IO_PROTOCOL_RESET_BUS ResetBus;
    EFI_SCSI_IO_PROTOCOL_RESET_DEVICE ResetDevice;
    EFI_SCSI_IO_PROTOCOL_EXECUTE_SCSI_COMMAND ExecuteScsiCommand;
    UINT32 IoAlign;
} EFI_SCSI_IO_PROTOCOL;
\end{verbatim}

**Parameters**

- **IoAlign**
  Supplies the alignment requirement for any buffer used in a data transfer. \texttt{IoAlign} values of 0 and 1 mean that the buffer can be placed anywhere in memory. Otherwise, \texttt{IoAlign} must be a power of 2, and the requirement is that the start address of a buffer must be evenly divisible by \texttt{IoAlign} with no remainder.

- **GetDeviceType**
  Retrieves the information of the device type which the SCSI device belongs to. See \texttt{GetDeviceType()}.  

- **GetDeviceLocation**
  Retrieves the device location information in the SCSI bus. See \texttt{GetDeviceLocation()}.  

- **ResetBus**
  Resets the entire SCSI bus the SCSI device attaches to. See \texttt{ResetBus()}.  

- **ResetDevice**
  Resets the SCSI Device that is specified by the device handle the SCSI I/O protocol attaches. See \texttt{ResetDevice()}.  

- **ExecuteScsiCommand**
  Sends a SCSI command to the SCSI device and waits for the execution completion until an exit condition is met, or a timeout occurs. See \texttt{ExecuteScsiCommand()}.  

**Description**

The \texttt{EFI_SCSI_IO_PROTOCOL} provides the basic functionalities to access and manage a SCSI Device. There is one \texttt{EFI_SCSI_IO_PROTOCOL} instance for each SCSI Device on a SCSI Bus. A device driver that wishes to manage a SCSI Device in a system will have to retrieve the \texttt{EFI_SCSI_IO_PROTOCOL} instance
that is associated with the SCSI Device. A device handle for a SCSI Device will minimally contain an
EFI_DEVICE_PATH_PROTOCOL instance and an EFI_SCSI_IO_PROTOCOL instance.

**EFI_SCSI_IO_PROTOCOL.GetDeviceType()**

**Summary**
Retrieves the device type information of the SCSI Device.

**Prototype**
```c
typedef EFI_STATUS
  (EFIAPICALL *EFI_SCSI_IO_PROTOCOL_GET_DEVICE_TYPE) (
    IN EFI_SCSI_IO_PROTOCOL   *This,
    OUT UINT8                 *DeviceType
  );
```

**Parameters**
- **This**
  A pointer to the EFI_SCSI_IO_PROTOCOL instance. Type
  EFI_SCSI_IO_PROTOCOL is defined in EFI_SCSI_IO_PROTOCOL.

- **DeviceType**
  A pointer to the device type information retrieved from the SCSI Device. See “Related Definitions” for the possible returned values of this parameter.

**Description**
This function is used to retrieve the SCSI device type information. This function is typically used for SCSI Device Drivers to quickly recognize whether the SCSI Device could be managed by it.

If **DeviceType** is NULL, then EFI_INVALID_PARAMETER is returned. Otherwise, the device type is returned in DeviceType and EFI_SUCCESS is returned.
Related Definitions

//Defined in the SCSI Primary Commands standard (e.g., SPC-4)

#define EFI_SCSI_IO_TYPE_DISK 0x00 // Disk device
#define EFI_SCSI_IO_TYPE_TAPE 0x01 // Tape device
#define EFI_SCSI_IO_TYPE_PRINTER 0x02 // Printer
#define EFI_SCSI_IO_TYPE_PROCESSOR 0x03 // Processor
#define EFI_SCSI_IO_TYPE_WORM 0x04 // Write-once read-multiple
#define EFI_SCSI_IO_TYPE_CDROM 0x05 // CD or DVD device
#define EFI_SCSI_IO_TYPE.Scanner 0x06 // Scanner device
#define EFI_SCSI_IO_TYPE_MEDIUMCHANGER 0x08 // Medium Changer device
#define EFI_SCSI_IO_TYPE_COMMUNICATION 0x09 // Communications device
#define MFI_SCSI_IO_TYPE_A 0x0A // Obsolete
#define MFI_SCSI_IO_TYPE_B 0x0B // Obsolete
#define MFI_SCSI_IO_TYPE_RAID 0x0C // Storage array controller
    // device (e.g., RAID)
#define MFI_SCSI_IO_TYPE_SES 0x0D // Enclosure services device
#define MFI_SCSI_IO_TYPE_RBC 0x0E // Simplified direct-access
    // device (e.g., magnetic
    // disk)
#define MFI_SCSI_IO_TYPE_OCRW 0x0F // Optical card reader/
    // writer device
#define MFI_SCSI_IO_TYPE_BRIDGE 0x10 // Bridge Controller
    // Commands
#define MFI_SCSI_IO_TYPE_OSD 0x11 // Object-based Storage
    // Device
#define EFI_SCSI_IO_TYPE RESERVED_LOW 0x12 // Reserved (low)
#define EFI_SCSI_IO_TYPE_RESERVED_HIGH 0x1E // Reserved (high)
#define EFI_SCSI_IO_TYPE_UNKNOWN 0x1F // Unknown no device type

Status Codes Returned

<table>
<thead>
<tr>
<th>Code</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>EFI_SUCCESS</td>
<td>Retrieves the device type information successfully.</td>
</tr>
<tr>
<td>EFI_INVALID_PARAMETER</td>
<td>The DeviceType is <strong>NULL</strong>.</td>
</tr>
</tbody>
</table>

**EFI_SCSI_IO_PROTOCOL.GetDeviceLocation()**

Summary

Retrieves the SCSI device location in the SCSI channel.
Prototype

typedef
EFI_STATUS
(EFIAPI *EFI_SCSI_IO_PROTOCOL_GET_DEVICE_LOCATION) (  
IN EFI_SCSI_IO_PROTOCOL *This,
IN OUT UINT8 **Target,
OUT UINT64 *Lun
);

Parameters

This A pointer to the EFI_SCSI_IO_PROTOCOL instance. Type EFI_SCSI_IO_PROTOCOL is defined in EFI_SCSI_IO_PROTOCOL.

Target A pointer to the Target Array which represents the ID of a SCSI device on the SCSI channel.

Lun A pointer to the Logical Unit Number of the SCSI device on the SCSI channel.

Description

This function is used to retrieve the SCSI device location in the SCSI bus. The device location is determined by a (Target, Lun) pair. This function allows a SCSI Device Driver to retrieve its location on the SCSI channel, and may use the Extended SCSI Pass Thru Protocol to access the SCSI device directly.

If Target or Lun is NULL, then EFI_INVALID_PARAMETER is returned. Otherwise, the device location is returned in Target and Lun, and EFI_SUCCESS is returned.

Status Codes Returned

<table>
<thead>
<tr>
<th>Code</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>EFI_SUCCESS</td>
<td>Retrieves the device location successfully.</td>
</tr>
<tr>
<td>EFI_INVALID_PARAMETER</td>
<td>Target or Lun is NULL</td>
</tr>
</tbody>
</table>

EFI_SCSI_IO_PROTOCOL.ResetBus()

Summary

Resets the SCSI Bus that the SCSI Device is attached to.

Prototype

typedef
EFI_STATUS
(EFIAPI *EFI_SCSI_IO_PROTOCOL_RESET_BUS) (  
IN EFI_SCSI_IO_PROTOCOL *This
);

Parameters

This A pointer to the EFI_SCSI_IO_PROTOCOL instance. Type EFI_SCSI_IO_PROTOCOL is defined in EFI_SCSI_IO_PROTOCOL.
Description
This function provides the mechanism to reset the whole SCSI bus that the specified SCSI Device is connected to. Some SCSI Host Controller may not support bus reset, if so, **EFI_UNSUPPORTED** is returned. If a device error occurs while executing that bus reset operation, then **EFI_DEVICE_ERROR** is returned. If a timeout occurs during the execution of the bus reset operation, then **EFI_TIMEOUT** is returned. If the bus reset operation is completed, then **EFI_SUCCESS** is returned.

Status Codes Returned

<table>
<thead>
<tr>
<th>Code</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>EFI_SUCCESS</td>
<td>The SCSI bus is reset successfully.</td>
</tr>
<tr>
<td>EFI_DEVICE_ERROR</td>
<td>Errors encountered when resetting the SCSI bus.</td>
</tr>
<tr>
<td>EFI_UNSUPPORTED</td>
<td>The bus reset operation is not supported by the SCSI Host Controller.</td>
</tr>
<tr>
<td>EFI_TIMEOUT</td>
<td>A timeout occurred while attempting to reset the SCSI bus.</td>
</tr>
</tbody>
</table>

**EFI_SCSI_IO_PROTOCOL.ResetDevice()**

Summary
Resets the SCSI Device that is specified by the device handle that the SCSI I/O Protocol is attached.

Prototype
```c
typedef EFI_STATUS (EFIAPI *EFI_SCSI_IO_PROTOCOL_RESET_DEVICE) (
    IN EFI_SCSI_IO_PROTOCOL *This
);
```

Parameters

| This            | A pointer to the **EFI_SCSI_IO_PROTOCOL** instance. Type **EFI_SCSI_IO_PROTOCOL** is defined in **EFI_SCSI_IO_PROTOCOL**. |

Description
This function provides the mechanism to reset the SCSI Device. If the SCSI bus does not support a device reset operation, then **EFI_UNSUPPORTED** is returned. If a device error occurs while executing that device reset operation, then **EFI_DEVICE_ERROR** is returned. If a timeout occurs during the execution of the device reset operation, then **EFI_TIMEOUT** is returned. If the device reset operation is completed, then **EFI_SUCCESS** is returned.
Status Codes Returned

<table>
<thead>
<tr>
<th>Code</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>EFI_SUCCESS</td>
<td>Reset the SCSI Device successfully.</td>
</tr>
<tr>
<td>EFI_DEVICE_ERROR</td>
<td>Errors are encountered when resetting the SCSI Device.</td>
</tr>
<tr>
<td>EFI_UNSUPPORTED</td>
<td>The SCSI bus does not support a device reset operation.</td>
</tr>
<tr>
<td>EFI_TIMEOUT</td>
<td>A timeout occurred while attempting to reset the SCSI Device.</td>
</tr>
</tbody>
</table>

EFI_SCSI_IO_PROTOCOL.ExecuteScsiCommand()

**Summary**
Sends a SCSI Request Packet to the SCSI Device for execution.

**Prototype**
```c
typedef EFIAPI *EFI_SCSI_IO_PROTOCOL_EXECUTE_SCSI_COMMAND) (  
    IN EFI_SCSI_IO_PROTOCOL *This,  
    IN OUT EFI_SCSI_IO_SCSI_REQUEST_PACKET Packet,  
    IN EFI_EVENT Event OPTIONAL  
);
```

**Parameters**
- **This**
  A pointer to the EFI_SCSI_IO_PROTOCOL instance. Type EFI_SCSI_IO_PROTOCOL is defined in EFI_SCSI_IO_PROTOCOL.
- **Packet**
  The SCSI request packet to send to the SCSI Device specified by the device handle. See “Related Definitions” for a description of EFI_SCSI_IO_SCSI_REQUEST_PACKET.
- **Event**
  If the SCSI bus where the SCSI device is attached does not support non-blocking I/O, then Event is ignored, and blocking I/O is performed. If Event is NULL, then blocking I/O is performed. If Event is not NULL and non-blocking I/O is supported, then non-blocking I/O is performed, and Event will be signaled when the SCSI Request Packet completes.
Related Definitions

typedef struct {
    UINT64   Timeout;
    VOID    *InDataBuffer;
    VOID    *OutDataBuffer;
    VOID    *SenseData;
    VOID    *Cdb;
    UINT32  InTransferLength;
    UINT32  OutTransferLength;
    UINT8   CdbLength;
    UINT8   DataDirection;
    UINT8   HostAdapterStatus;
    UINT8   TargetStatus;
    UINT8   SenseDataLength;
} EFI_SCSI_IO_SCSI_REQUEST_PACKET;

Timeout
The timeout, in 100 ns units, to use for the execution of this SCSI Request Packet. A Timeout value of 0 means that this function will wait indefinitely for the SCSI Request Packet to execute. If Timeout is greater than zero, then this function will return EFI_TIMEOUT if the time required to execute the SCSI Request Packet is greater than Timeout.

DataBuffer
A pointer to the data buffer to transfer from or to the SCSI device.

InDataBuffer
A pointer to the data buffer to transfer between the SCSI controller and the SCSI device for SCSI READ command. For all SCSI WRITE Commands this must point to NULL.

OutDataBuffer
A pointer to the data buffer to transfer between the SCSI controller and the SCSI device for SCSI WRITE command. For all SCSI READ commands this field must point to NULL.

SenseData
A pointer to the sense data that was generated by the execution of the SCSI Request Packet.

Cdb
A pointer to buffer that contains the Command Data Block to send to the SCSI device.

InTransferLength
On Input, the size, in bytes, of InDataBuffer. On output, the number of bytes transferred between the SCSI controller and the SCSI device. If InTransferLength is larger than the SCSI controller can handle, no data will be transferred, InTransferLength will be updated to contain the number of bytes that the SCSI controller is able to transfer, and EFI_BAD_BUFFER_SIZE will be returned.

OutTransferLength
On Input, the size, in bytes of OutDataBuffer. On Output, the Number of bytes transferred between SCSI Controller and the SCSI device. If OutTransferLength is larger than the SCSI controller can handle, no data will be transferred, OutTransferLength will be updated to contain the number of bytes that the SCSI controller is able to transfer, and EFI_BAD_BUFFER_SIZE will be returned.
CdbLength

The length, in bytes, of the buffer Cdb. The standard values are 6, 10, 12, and 16, but other values are possible if a variable length CDB is used.

DataDirection

The direction of the data transfer. 0 for reads, 1 for writes. A value of 2 is Reserved for Bi-Directional SCSI commands. For example XDREADWRITE. All other values are reserved, and must not be used.

HostAdapterStatus

The status of the SCSI Host Controller that produces the SCSI bus where the SCSI device attached when the SCSI Request Packet was executed on the SCSI Controller. See the possible values listed below.

TargetStatus

The status returned by the SCSI device when the SCSI Request Packet was executed. See the possible values listed below.

SenseDataLength

On input, the length in bytes of the SenseData buffer. On output, the number of bytes written to the SenseData buffer.

//
// DataDirection
//
#define EFI_SCSI_IO_DATA_DIRECTION_READ           0
#define EFI_SCSI_IO_DATA_DIRECTION_WRITE          1
#define EFI_SCSI_IO_DATA_DIRECTION_BIDIRECTIONAL  2

//
// HostAdapterStatus
//
#define EFI_SCSI_IO_STATUS_HOST_ADAPTER_OK                    0x00
#define EFI_SCSI_IO_STATUS_HOST_ADAPTER_TIMEOUT_COMMAND       0x09
#define EFI_SCSI_IO_STATUS_HOST_ADAPTER_TIMEOUT              0x0b
#define EFI_SCSI_IO_STATUS_HOST_ADAPTER_MESSAGE_REJECT        0x0d
#define EFI_SCSI_IO_STATUS_HOST_ADAPTER_BUS_RESET            0x0e
#define EFI_SCSI_IO_STATUS_HOST_ADAPTER_PARITY_ERROR          0x0f
#define EFI_SCSI_IO_STATUS_HOST_ADAPTER_REQUESTSENSE_FAILED   0x10
#define EFI_SCSI_IO_STATUS_HOST_ADAPTER_SELECTION_TIMEOUT     0x11
#define EFI_SCSI_IO_STATUS_HOST_ADAPTER_DATAOVERRUNUNDERRUN  0x12
#define EFI_SCSI_IO_STATUS_HOST_ADAPTER_BUSFREE               0x13
#define EFI_SCSI_IO_STATUS_HOST_ADAPTER_PHASE_ERROR           0x14
#define EFI_SCSI_IO_STATUS_HOST_ADAPTER_OTHER                 0x7f

//
// TargetStatus
//
#define EFI_SCSI_IO_STATUS_TARGET_GOOD                        0x00
#define EFI_SCSI_IO_STATUS_TARGET_CHECK_CONDITION             0x02
#define EFI_SCSI_IO_STATUS_TARGETCONDITIONMET                 0x04
#define EFI_SCSI_IO_STATUS_TARGETBUSY                         0x08
#define EFI_SCSI_IO_STATUSARGETINTERMEDIATE                  0x10
#define EFI_SCSI_IO_STATUS_TARGET_INTERMEDIATE_CONDITION_MET 0x14
#define EFI_SCSI_IO_STATUS_TARGET_RESERVATION_CONFLICT 0x18
#define EFI_SCSI_IO_STATUS_TARGET_COMMAND_TERMINATED 0x22
#define EFI_SCSI_IO_STATUS_TARGET_QUEUE_FULL 0x28

**Description**

This function sends the SCSI Request Packet specified by `Packet` to the SCSI Device.

If the SCSI Bus supports non-blocking I/O and `Event` is not `NULL`, then this function will return immediately after the command is sent to the SCSI Device, and will later signal `Event` when the command has completed. If the SCSI Bus supports non-blocking I/O and `Event` is `NULL`, then this function will send the command to the SCSI Device and block until it is complete. If the SCSI Bus does not support non-blocking I/O, the `Event` parameter is ignored, and the function will send the command to the SCSI Device and block until it is complete.

If `Packet` is successfully sent to the SCSI Device, then `EFI_SUCCESS` is returned.

If `Packet` cannot be sent because there are too many packets already queued up, then `EFI_NOT_READY` is returned. The caller may retry `Packet` at a later time.

If a device error occurs while sending the `Packet`, then `EFI_DEVICE_ERROR` is returned.

If a timeout occurs during the execution of `Packet`, then `EFI_TIMEOUT` is returned.

If any field of `Packet` is invalid, then `EFI_INVALID_PARAMETER` is returned.

If the data buffer described by `DataBuffer` and `TransferLength` is too big to be transferred in a single command, then `EFI_BAD_BUFFER_SIZE` is returned. The number of bytes actually transferred is returned in `TransferLength`.

If the command described in `Packet` is not supported by the SCSI Host Controller that produces the SCSI bus, then `EFI_UNSUPPORTED` is returned.

If `EFI_SUCCESS`, `EFI_BAD_BUFFER_SIZE`, `EFI_DEVICE_ERROR`, or `EFI_TIMEOUT` is returned, then the caller must examine the status fields in `Packet` in the following precedence order: `HostAdapterStatus` followed by `TargetStatus` followed by `SenseDataLength`, followed by `SenseData`. If non-blocking I/O is being used, then the status fields in `Packet` will not be valid until the `Event` associated with `Packet` is signaled.

If `EFI_NOT_READY`, `EFI_INVALID_PARAMETER` or `EFI_UNSUPPORTED` is returned, then `Packet` was never sent, so the status fields in `Packet` are not valid. If non-blocking I/O is being used, the `Event` associated with `Packet` will not be signaled.

**Status Codes Returned**

<table>
<thead>
<tr>
<th>Code</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>EFI_SUCCESS</td>
<td>The SCSI Request Packet was sent by the host. For read and bi-directional commands, <code>InTransferLength</code> bytes were transferred to <code>InDataBuffer</code>. For write and bi-directional commands, <code>OutTransferLength</code> bytes were transferred from <code>OutDataBuffer</code>. See <code>HostAdapterStatus</code>, <code>TargetStatus</code>, <code>SenseDataLength</code>, and <code>SenseData</code> in that order for additional status information.</td>
</tr>
</tbody>
</table>
15.5 SCSI Device Paths

An **EFI_SCSI_IO_PROTOCOL** must be installed on a handle for its services to be available to SCSI device drivers. In addition to the **EFI_SCSI_IO_PROTOCOL**, an **EFI_DEVICE_PATH_PROTOCOL** must also be installed on the same handle. See Section 10 for detailed description of the **EFI_DEVICE_PATH_PROTOCOL**.

The SCSI Driver Model defined in this document can support the SCSI channel generated or emulated by multiple architectures, such as Parallel SCSI, ATAPI, Fibre Channel, InfiniBand, and other future channel types. In this section, there are four example device paths provided, including SCSI device path, ATAPI device path, Fibre Channel device path and InfiniBand device path.

### 15.5.1 SCSI Device Path Example

Table 15-1 shows an example device path for a SCSI device controller on a desktop platform. This SCSI device controller is connected to a SCSI channel that is generated by a PCI SCSI host controller. The PCI SCSI host controller generates a single SCSI channel, it is located at PCI device number 0x07 and PCI function 0x00, and is directly attached to a PCI root bridge. The SCSI device controller is assigned SCSI Id 2, and its LUN is 0.

<table>
<thead>
<tr>
<th>Code</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>EFI_BAD_BUFFER_SIZE</strong></td>
<td>The SCSI Request Packet was not executed. For read and bi-directional commands, the number of bytes that could be transferred is returned in <strong>InTransferLength</strong>. For write and bi-directional commands, the number of bytes that could be transferred is returned in <strong>OutTransferLength</strong>. See <strong>HostAdapterStatus</strong> and <strong>TargetStatus</strong> in that order for additional status information.</td>
</tr>
<tr>
<td><strong>EFI_NOT_READY</strong></td>
<td>The SCSI Request Packet could not be sent because there are too many SCSI Command Packets already queued. The caller may retry again later.</td>
</tr>
<tr>
<td><strong>EFI_DEVICE_ERROR</strong></td>
<td>A device error occurred while attempting to send the SCSI Request Packet. See <strong>HostAdapterStatus</strong>, <strong>TargetStatus</strong>, <strong>SenseDataLength</strong>, and <strong>SenseData</strong> in that order for additional status information.</td>
</tr>
<tr>
<td><strong>EFI_INVALID_PARAMETER</strong></td>
<td>The contents of <strong>CommandPacket</strong> are invalid. The SCSI Request Packet was not sent, so no additional status information is available.</td>
</tr>
<tr>
<td><strong>EFI_UNSUPPORTED</strong></td>
<td>The command described by the SCSI Request Packet is not supported by the SCSI initiator (i.e., SCSI Host Controller). The SCSI Request Packet was not sent, so no additional status information is available.</td>
</tr>
<tr>
<td><strong>EFI_TIMEOUT</strong></td>
<td>A timeout occurred while waiting for the SCSI Request Packet to execute. See <strong>HostAdapterStatus</strong>, <strong>TargetStatus</strong>, <strong>SenseDataLength</strong>, and <strong>SenseData</strong> in that order for additional status information.</td>
</tr>
</tbody>
</table>
This sample device path consists of an ACPI Device Path Node, a PCI Device Path Node, a SCSI Node, and a Device Path End Structure. The _HID and _UID must match the ACPI table description of the PCI Root Bridge. The shorthand notation for this device path is:

\[ \text{ACPI(PNP0A03,0)/PCI(7,0)/SCSI(2,0)}. \]

### Table 15-1 SCSI Device Path Examples

<table>
<thead>
<tr>
<th>Byte Offset</th>
<th>Byte Length</th>
<th>Data</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x00</td>
<td>0x01</td>
<td>0x02</td>
<td>Generic Device Path Header – Type ACPI Device Path</td>
</tr>
<tr>
<td>0x01</td>
<td>0x01</td>
<td>0x01</td>
<td>Sub type – ACPI Device Path</td>
</tr>
<tr>
<td>0x02</td>
<td>0x02</td>
<td>0x0C</td>
<td>Length – 0x0C bytes</td>
</tr>
<tr>
<td>0x04</td>
<td>0x04</td>
<td>0x41D0, 0x0A03</td>
<td>_HID PNP0A03 – 0x41D0 represents the compressed string ‘PNP’ and is encoded in the low order bytes. The compression method is described in the ACPI Specification.</td>
</tr>
<tr>
<td>0x08</td>
<td>0x04</td>
<td>0x0000</td>
<td>_UID</td>
</tr>
<tr>
<td>0x0C</td>
<td>0x01</td>
<td>0x01</td>
<td>Generic Device Path Header – Type Hardware Device Path</td>
</tr>
<tr>
<td>0x0D</td>
<td>0x01</td>
<td>0x01</td>
<td>Sub type – PCI</td>
</tr>
<tr>
<td>0x0E</td>
<td>0x02</td>
<td>0x06</td>
<td>Length – 0x06 bytes</td>
</tr>
<tr>
<td>0x10</td>
<td>0x01</td>
<td>0x07</td>
<td>PCI Function</td>
</tr>
<tr>
<td>0x11</td>
<td>0x01</td>
<td>0x00</td>
<td>PCI Device</td>
</tr>
<tr>
<td>0x12</td>
<td>0x01</td>
<td>0x03</td>
<td>Generic Device Path Header – Type Message Device Path</td>
</tr>
<tr>
<td>0x13</td>
<td>0x01</td>
<td>0x02</td>
<td>Sub type – SCSI</td>
</tr>
<tr>
<td>0x14</td>
<td>0x02</td>
<td>0x08</td>
<td>Length – 0x08 bytes</td>
</tr>
<tr>
<td>0x16</td>
<td>0x02</td>
<td>0x0002</td>
<td>Target ID on the SCSI bus (PUN)</td>
</tr>
<tr>
<td>0x18</td>
<td>0x02</td>
<td>0x0000</td>
<td>Logical Unit Number (LUN)</td>
</tr>
<tr>
<td>0x1A</td>
<td>0x01</td>
<td>0xff</td>
<td>Generic Device Path Header – Type End of Hardware Device Path</td>
</tr>
<tr>
<td>0x1B</td>
<td>0x01</td>
<td>0xFF</td>
<td>Sub type – End of Entire Device Path</td>
</tr>
<tr>
<td>0x1C</td>
<td>0x02</td>
<td>0x04</td>
<td>Length – 0x04 bytes</td>
</tr>
</tbody>
</table>

### 15.5.2 ATAPI Device Path Example

Table 15-2 shows an example device path for an ATAPI device on a desktop platform. This ATAPI device is connected to the IDE bus on Primary channel, and is configured as the Master device on the channel. The IDE bus is generated by the IDE controller that is a PCI device. It is located at PCI device number 0x1F and PCI function 0x01, and is directly attached to a PCI root bridge.

This sample device path consists of an ACPI Device Path Node, a PCI Device Path Node, an ATAPI Node, and a Device Path End Structure. The _HID and _UID must match the ACPI table description of the PCI Root Bridge. The shorthand notation for this device path is:

\[ \text{ACPI(PNP0A03,0)/PCI(7,0)/ATA(Primary,Master,0)}. \]
Table 15-2 ATAPI Device Path Examples

<table>
<thead>
<tr>
<th>Byte Offset</th>
<th>Byte Length</th>
<th>Data</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x00</td>
<td>0x01</td>
<td>0x02</td>
<td>Generic Device Path Header – Type ACPI Device Path</td>
</tr>
<tr>
<td>0x01</td>
<td>0x01</td>
<td>0x01</td>
<td>Sub type – ACPI Device Path</td>
</tr>
<tr>
<td>0x02</td>
<td>0x02</td>
<td>0x0C</td>
<td>Length – 0x0C bytes</td>
</tr>
<tr>
<td>0x04</td>
<td>0x04</td>
<td>0x41D0, 0xA03</td>
<td>_HID PNP0A03 – 0x41D0 represents the compressed string ‘PNP’ and is encoded in the low order bytes. The compression method is described in the ACPI Specification.</td>
</tr>
<tr>
<td>0x08</td>
<td>0x04</td>
<td>0x0000</td>
<td>_UID</td>
</tr>
<tr>
<td>0x0C</td>
<td>0x01</td>
<td>0x01</td>
<td>Generic Device Path Header – Type Hardware Device Path</td>
</tr>
<tr>
<td>0x0D</td>
<td>0x01</td>
<td>0x01</td>
<td>Sub type – PCI</td>
</tr>
<tr>
<td>0x0E</td>
<td>0x02</td>
<td>0x06</td>
<td>Length – 0x06 bytes</td>
</tr>
<tr>
<td>0x10</td>
<td>0x01</td>
<td>0x07</td>
<td>PCI Function</td>
</tr>
<tr>
<td>0x11</td>
<td>0x01</td>
<td>0x00</td>
<td>PCI Device</td>
</tr>
<tr>
<td>0x12</td>
<td>0x01</td>
<td>0x03</td>
<td>Generic Device Path Header – Type Message Device Path</td>
</tr>
<tr>
<td>0x13</td>
<td>0x01</td>
<td>0x01</td>
<td>Sub type – ATAPI</td>
</tr>
<tr>
<td>0x14</td>
<td>0x02</td>
<td>0x08</td>
<td>Length – 0x08 bytes</td>
</tr>
<tr>
<td>0x16</td>
<td>0x01</td>
<td>0x00</td>
<td>PrimarySecondary – Set to zero for primary or one for secondary.</td>
</tr>
<tr>
<td>0x17</td>
<td>0x01</td>
<td>0x00</td>
<td>SlaveMaster – set to zero for master or one for slave.</td>
</tr>
<tr>
<td>0x18</td>
<td>0x02</td>
<td>0x0000</td>
<td>Logical Unit Number, LUN.</td>
</tr>
<tr>
<td>0x1A</td>
<td>0x01</td>
<td>0xFF</td>
<td>Generic Device Path Header – Type End of Hardware Device Path</td>
</tr>
<tr>
<td>0x1B</td>
<td>0x01</td>
<td>0xFF</td>
<td>Sub type – End of Entire Device Path</td>
</tr>
<tr>
<td>0x1C</td>
<td>0x02</td>
<td>0x04</td>
<td>Length – 0x04 bytes</td>
</tr>
</tbody>
</table>

15.5.3 Fibre Channel Device Path Example

Table 15-3 shows an example device path for an SCSI device that is connected to a Fibre Channel Port on a desktop platform. The Fibre Channel Port is a PCI device that is located at PCI device number 0x08 and PCI function 0x00, and is directly attached to a PCI root bridge. The Fibre Channel Port is addressed by the World Wide Number, and is assigned as X (X is a 64bit value); the SCSI device’s Logical Unit Number is 0.

This sample device path consists of an ACPI Device Path Node, a PCI Device Path Node, a Fibre Channel Device Path Node, and a Device Path End Structure. The _HID and _UID must match the ACPI table description of the PCI Root Bridge. The shorthand notation for this device path is:

\[
\text{ACPI(PNP0A03, 0)/PCI(8, 0)/Fibre(X, 0)}.\]
Table 15-3 Fibre Channel Device Path Examples

<table>
<thead>
<tr>
<th>Byte Offset</th>
<th>Byte Length</th>
<th>Data</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x00</td>
<td>0x01</td>
<td>0x02</td>
<td>Generic Device Path Header – Type ACPI Device Path</td>
</tr>
<tr>
<td>0x01</td>
<td>0x01</td>
<td>0x01</td>
<td>Sub type – ACPI Device Path</td>
</tr>
<tr>
<td>0x02</td>
<td>0x02</td>
<td>0x0C</td>
<td>Length – 0x0C bytes</td>
</tr>
<tr>
<td>0x04</td>
<td>0x04</td>
<td>0x41D0, 0x0A03</td>
<td>_HID PNP0A03 – 0x41D0 represents the compressed string ‘PNP’ and is encoded in the low order bytes. The compression method is described in the ACPI Specification.</td>
</tr>
<tr>
<td>0x08</td>
<td>0x04</td>
<td>0x0000</td>
<td>_UID</td>
</tr>
<tr>
<td>0x0C</td>
<td>0x01</td>
<td>0x01</td>
<td>Generic Device Path Header – Type Hardware Device Path</td>
</tr>
<tr>
<td>0x0D</td>
<td>0x01</td>
<td>0x01</td>
<td>Sub type – PCI</td>
</tr>
<tr>
<td>0x0E</td>
<td>0x02</td>
<td>0x06</td>
<td>Length – 0x06 bytes</td>
</tr>
<tr>
<td>0x10</td>
<td>0x01</td>
<td>0x08</td>
<td>PCI Function</td>
</tr>
<tr>
<td>0x11</td>
<td>0x01</td>
<td>0x00</td>
<td>PCI Device</td>
</tr>
<tr>
<td>0x12</td>
<td>0x01</td>
<td>0x03</td>
<td>Generic Device Path Header – Type Message Device Path</td>
</tr>
<tr>
<td>0x13</td>
<td>0x01</td>
<td>0x02</td>
<td>Sub type – Fibre Channel</td>
</tr>
<tr>
<td>0x14</td>
<td>0x02</td>
<td>0x24</td>
<td>Length – 0x24 bytes</td>
</tr>
<tr>
<td>0x16</td>
<td>0x04</td>
<td>0x00</td>
<td>Reserved</td>
</tr>
<tr>
<td>0x1A</td>
<td>0x08</td>
<td>X Fibre Channel World Wide Number</td>
<td></td>
</tr>
<tr>
<td>0x22</td>
<td>0x08</td>
<td>0x00</td>
<td>Fibre Channel Logical Unit Number (LUN).</td>
</tr>
<tr>
<td>0x2A</td>
<td>0x01</td>
<td>0xFF</td>
<td>Generic Device Path Header – Type End of Hardware Device Path</td>
</tr>
<tr>
<td>0x2B</td>
<td>0x01</td>
<td>0xFF</td>
<td>Sub type – End of Entire Device Path</td>
</tr>
<tr>
<td>0x2C</td>
<td>0x02</td>
<td>0x04</td>
<td>Length – 0x04 bytes</td>
</tr>
</tbody>
</table>

15.5.4 InfiniBand Device Path Example

Table 15-4 shows an example device path for a SCSI device in an InfiniBand Network. This SCSI device is connected to a single SCSI channel generated by a SCS Host Adapter, and the SCSI Host Adapter is an end node in the InfiniBand Network. The SCSI Host Adapter is a PCI device that is located at PCI device number 0x07 and PCI function 0x00, and is directly attached to a PCI root bridge. The SCSI device is addressed by the (IOU X, IOC Y, Deviceld Z) in the InfiniBand Network. (X, Y, Z are EUI-64 compliant identifiers).

This sample device path consists of an ACPI Device Path Node, a PCI Device Path Node, an InfiniBand Node, and a Device Path End Structure. The _HID and _UID must match the ACPI table description of the PCI Root Bridge. The shorthand notation for this device path is:

ACPI(PNP0A03, 0)/PCI(7, 0)/Infiniband(X,Y,Z).
Table 15-4 InfiniBand Device Path Examples

<table>
<thead>
<tr>
<th>Byte Offset</th>
<th>Byte Length</th>
<th>Data</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x00</td>
<td>0x01</td>
<td>0x02</td>
<td>Generic Device Path Header – Type ACPI Device Path</td>
</tr>
<tr>
<td>0x01</td>
<td>0x01</td>
<td>0x01</td>
<td>Sub type – ACPI Device Path</td>
</tr>
<tr>
<td>0x02</td>
<td>0x02</td>
<td>0x0C</td>
<td>Length – 0x0C bytes</td>
</tr>
<tr>
<td>0x04</td>
<td>0x04</td>
<td>0x41D0, 0x0A03</td>
<td>_HID PNP0A03 – 0x41D0 represents the compressed string ‘PNP’ and is encoded in the low order bytes. The compression method is described in the ACPI Specification.</td>
</tr>
<tr>
<td>0x08</td>
<td>0x04</td>
<td>0x0000</td>
<td>_UID</td>
</tr>
<tr>
<td>0x0C</td>
<td>0x01</td>
<td>0x01</td>
<td>Generic Device Path Header – Type Hardware Device Path</td>
</tr>
<tr>
<td>0x0D</td>
<td>0x01</td>
<td>0x01</td>
<td>Sub type – PCI</td>
</tr>
<tr>
<td>0x0E</td>
<td>0x02</td>
<td>0x06</td>
<td>Length – 0x06 bytes</td>
</tr>
<tr>
<td>0x10</td>
<td>0x01</td>
<td>0x07</td>
<td>PCI Function</td>
</tr>
<tr>
<td>0x11</td>
<td>0x01</td>
<td>0x00</td>
<td>PCI Device</td>
</tr>
<tr>
<td>0x12</td>
<td>0x01</td>
<td>0x03</td>
<td>Generic Device Path Header – Type Message Device Path</td>
</tr>
<tr>
<td>0x13</td>
<td>0x01</td>
<td>0x09</td>
<td>Sub type – InfiniBand</td>
</tr>
<tr>
<td>0x14</td>
<td>0x02</td>
<td>0x20</td>
<td>Length – 0x20 bytes</td>
</tr>
<tr>
<td>0x16</td>
<td>0x04</td>
<td>0x00</td>
<td>Reserved</td>
</tr>
<tr>
<td>0x1A</td>
<td>0x08</td>
<td>X</td>
<td>64bit node GUID of the IOU</td>
</tr>
<tr>
<td>0x22</td>
<td>0x08</td>
<td>Y</td>
<td>64bit GUID of the IOC</td>
</tr>
<tr>
<td>0x2A</td>
<td>0x08</td>
<td>Z</td>
<td>64bit persistent ID of the device.</td>
</tr>
<tr>
<td>0x32</td>
<td>0x01</td>
<td>0xFF</td>
<td>Generic Device Path Header – Type End of Hardware Device Path</td>
</tr>
<tr>
<td>0x33</td>
<td>0x01</td>
<td>0xFF</td>
<td>Sub type – End of Entire Device Path</td>
</tr>
<tr>
<td>0x34</td>
<td>0x02</td>
<td>0x04</td>
<td>Length – 0x04 bytes</td>
</tr>
</tbody>
</table>

15.6 SCSI Pass Thru Device Paths

An EFI_EXT_SCSI_PASS_THRU_PROTOCOL must be installed on a handle for its services to be available to UEFI drivers and applications. In addition to the EFI_EXT_SCSI_PASS_THRU_PROTOCOL, an EFI_DEVICE_PATH_PROTOCOL must also be installed on the same handle. See Section 10 for a detailed description of the EFI_DEVICE_PATH_PROTOCOL.

A device path describes the location of a hardware component in a system from the processor’s point of view. This includes the list of busses that lie between the processor and the SCSI controller. The EFI Specification takes advantage of the ACPI Specification to name system components. For the following set of examples, a PCI SCSI controller is assumed. The examples will show a SCSI controller on the root PCI bus, and a SCSI controller behind a PCI-PCI bridge. In addition, an example of a multichannel SCSI controller will be shown.

Table 15-5 shows an example device path for a single channel PCI SCSI controller that is located at PCI device number 0x07 and PCI function 0x00, and is directly attached to a PCI root bridge. This device path consists of an ACPI Device Path Node, a PCI Device Path Node, and a Device Path End Structure. The _HID
and _UID must match the ACPI table description of the PCI Root Bridge. The shorthand notation for this device path is:

\[ \text{ACPI(PNP0A03,0)/PCI(7,0)}. \]

<table>
<thead>
<tr>
<th>Byte Offset</th>
<th>Byte Length</th>
<th>Data</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x00</td>
<td>0x01</td>
<td>0x02</td>
<td>Generic Device Path Header – Type ACPI Device Path</td>
</tr>
<tr>
<td>0x01</td>
<td>0x01</td>
<td>0x01</td>
<td>Sub type – ACPI Device Path</td>
</tr>
<tr>
<td>0x02</td>
<td>0x02</td>
<td>0x0C</td>
<td>Length – 0x0C bytes</td>
</tr>
<tr>
<td>0x04</td>
<td>0x04</td>
<td>0x41D0, 0x0A03</td>
<td>_HID PNP0A03 – 0x41D0 represents the compressed string ‘PNP’ and is encoded in the low order bytes. The compression method is described in the ACPI Specification.</td>
</tr>
<tr>
<td>0x08</td>
<td>0x04</td>
<td>0x0000</td>
<td>_UID</td>
</tr>
<tr>
<td>0x0C</td>
<td>0x01</td>
<td>0x01</td>
<td>Generic Device Path Header – Type Hardware Device Path</td>
</tr>
<tr>
<td>0x0D</td>
<td>0x01</td>
<td>0x01</td>
<td>Sub type – PCI</td>
</tr>
<tr>
<td>0x0E</td>
<td>0x02</td>
<td>0x06</td>
<td>Length – 0x06 bytes</td>
</tr>
<tr>
<td>0x10</td>
<td>0x01</td>
<td>0x00</td>
<td>PCI Function</td>
</tr>
<tr>
<td>0x11</td>
<td>0x01</td>
<td>0x07</td>
<td>PCI Device</td>
</tr>
<tr>
<td>0x12</td>
<td>0x01</td>
<td>0xFF</td>
<td>Generic Device Path Header – Type End of Hardware Device Path</td>
</tr>
<tr>
<td>0x13</td>
<td>0x01</td>
<td>0xFF</td>
<td>Sub type – End of Entire Device Path</td>
</tr>
<tr>
<td>0x14</td>
<td>0x02</td>
<td>0x04</td>
<td>Length – 0x04 bytes</td>
</tr>
</tbody>
</table>

Table 15-6 shows an example device path for a single channel PCI SCSI controller that is located behind a PCI to PCI bridge at PCI device number 0x07 and PCI function 0x00. The PCI to PCI bridge is directly attached to a PCI root bridge, and it is at PCI device number 0x05 and PCI function 0x00. This device path consists of an ACPI Device Path Node, two PCI Device Path Nodes, and a Device Path End Structure. The _HID and _UID must match the ACPI table description of the PCI Root Bridge. The shorthand notation for this device path is:

\[ \text{ACPI(PNP0A03,0)/PCI(5,0)/PCI(7,0)}. \]
Table 15-6 Single Channel PCI SCSI Controller behind a PCI Bridge

<table>
<thead>
<tr>
<th>Byte Offset</th>
<th>Byte Length</th>
<th>Data</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x00</td>
<td>0x01</td>
<td>0x02</td>
<td>Generic Device Path Header – Type ACPI Device Path</td>
</tr>
<tr>
<td>0x01</td>
<td>0x01</td>
<td>0x01</td>
<td>Sub type – ACPI Device Path</td>
</tr>
<tr>
<td>0x02</td>
<td>0x02</td>
<td>0x0C</td>
<td>Length – 0x0C bytes</td>
</tr>
<tr>
<td>0x04</td>
<td>0x04</td>
<td>0x41D0, 0xA03</td>
<td>_HID PNP0A03 – 0x41D0 represents the compressed string ‘PNP’ and is encoded in the low order bytes. The compression method is described in the ACPI Specification.</td>
</tr>
<tr>
<td>0x08</td>
<td>0x04</td>
<td>0x0000</td>
<td>_UID</td>
</tr>
<tr>
<td>0x0C</td>
<td>0x01</td>
<td>0x01</td>
<td>Generic Device Path Header – Type Hardware Device Path</td>
</tr>
<tr>
<td>0x0D</td>
<td>0x01</td>
<td>0x01</td>
<td>Sub type – PCI</td>
</tr>
<tr>
<td>0x0E</td>
<td>0x02</td>
<td>0x06</td>
<td>Length – 0x06 bytes</td>
</tr>
<tr>
<td>0x10</td>
<td>0x01</td>
<td>0x00</td>
<td>PCI Function</td>
</tr>
<tr>
<td>0x11</td>
<td>0x01</td>
<td>0x05</td>
<td>PCI Device</td>
</tr>
<tr>
<td>0x12</td>
<td>0x01</td>
<td>0x01</td>
<td>Generic Device Path Header – Type Hardware Device Path</td>
</tr>
<tr>
<td>0x13</td>
<td>0x01</td>
<td>0x01</td>
<td>Sub type – PCI</td>
</tr>
<tr>
<td>0x14</td>
<td>0x02</td>
<td>0x06</td>
<td>Length – 0x06 bytes</td>
</tr>
<tr>
<td>0x16</td>
<td>0x01</td>
<td>0x00</td>
<td>PCI Function</td>
</tr>
<tr>
<td>0x17</td>
<td>0x01</td>
<td>0x07</td>
<td>PCI Device</td>
</tr>
<tr>
<td>0x18</td>
<td>0x01</td>
<td>0xFF</td>
<td>Generic Device Path Header – Type End of Hardware Device Path</td>
</tr>
<tr>
<td>0x19</td>
<td>0x01</td>
<td>0xFF</td>
<td>Sub type – End of Entire Device Path</td>
</tr>
<tr>
<td>0x1A</td>
<td>0x02</td>
<td>0x04</td>
<td>Length – 0x04 bytes</td>
</tr>
</tbody>
</table>

Table 15-7 shows an example device path for channel #3 of a four channel PCI SCSI controller that is located behind a PCI to PCI bridge at PCI device number 0x07 and PCI function 0x00. The PCI to PCI bridge is directly attached to a PCI root bridge, and it is at PCI device number 0x05 and PCI function 0x00. This device path consists of an ACPI Device Path Node, two PCI Device Path Nodes, a Controller Node, and a Device Path End Structure. The _HID and _UID must match the ACPI table description of the PCI Root Bridge. The shorthand notation of the device paths for all four of the SCSI channels are listed below. Table 15-7 shows the last device path listed.

ACPI(PNP0A03,0)/PCI(5,0)/PCI(7,0)/Ctrl(0)
ACPI(PNP0A03,0)/PCI(5,0)/PCI(7,0)/Ctrl(1)
ACPI(PNP0A03,0)/PCI(5,0)/PCI(7,0)/Ctrl(2)
ACPI(PNP0A03,0)/PCI(5,0)/PCI(7,0)/Ctrl(3)
Table 15-7 Channel #3 of a PCI SCSI Controller behind a PCI Bridge

<table>
<thead>
<tr>
<th>Byte Offset</th>
<th>Byte Length</th>
<th>Data</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x00</td>
<td>0x01</td>
<td>0x02</td>
<td>Generic Device Path Header – Type ACPI Device Path</td>
</tr>
<tr>
<td>0x01</td>
<td>0x01</td>
<td>0x01</td>
<td>Sub type – ACPI Device Path</td>
</tr>
<tr>
<td>0x02</td>
<td>0x02</td>
<td>0x0C</td>
<td>Length – 0x0C bytes</td>
</tr>
<tr>
<td>0x04</td>
<td>0x04</td>
<td>0x41D0, 0x0A03</td>
<td>_HID PNP0A03 – 0x41D0 represents the compressed string ‘PNP’ and is encoded in the low order bytes. The compression method is described in the ACPI Specification.</td>
</tr>
<tr>
<td>0x08</td>
<td>0x04</td>
<td>0x0000</td>
<td>_UID</td>
</tr>
<tr>
<td>0x0C</td>
<td>0x01</td>
<td>0x01</td>
<td>Generic Device Path Header – Type Hardware Device Path</td>
</tr>
<tr>
<td>0x0D</td>
<td>0x01</td>
<td>0x01</td>
<td>Sub type – PCI</td>
</tr>
<tr>
<td>0x0E</td>
<td>0x02</td>
<td>0x06</td>
<td>Length – 0x06 bytes</td>
</tr>
<tr>
<td>0x10</td>
<td>0x01</td>
<td>0x00</td>
<td>PCI Function</td>
</tr>
<tr>
<td>0x11</td>
<td>0x01</td>
<td>0x05</td>
<td>PCI Device</td>
</tr>
<tr>
<td>0x12</td>
<td>0x01</td>
<td>0x01</td>
<td>Generic Device Path Header – Type Hardware Device Path</td>
</tr>
<tr>
<td>0x13</td>
<td>0x01</td>
<td>0x01</td>
<td>Sub type – PCI</td>
</tr>
<tr>
<td>0x14</td>
<td>0x02</td>
<td>0x06</td>
<td>Length – 0x06 bytes</td>
</tr>
<tr>
<td>0x16</td>
<td>0x01</td>
<td>0x00</td>
<td>PCI Function</td>
</tr>
<tr>
<td>0x17</td>
<td>0x01</td>
<td>0x07</td>
<td>PCI Device</td>
</tr>
<tr>
<td>0x18</td>
<td>0x01</td>
<td>0x01</td>
<td>Generic Device Path Header – Type Hardware Device Path</td>
</tr>
<tr>
<td>0x19</td>
<td>0x01</td>
<td>0x05</td>
<td>Sub type – Controller</td>
</tr>
<tr>
<td>0x1A</td>
<td>0x02</td>
<td>0x08</td>
<td>Length – 0x08 bytes</td>
</tr>
<tr>
<td>0x1C</td>
<td>0x04</td>
<td>0x0003</td>
<td>Controller Number</td>
</tr>
<tr>
<td>0x20</td>
<td>0x01</td>
<td>0xFF</td>
<td>Generic Device Path Header – Type End of Hardware Device Path</td>
</tr>
<tr>
<td>0x21</td>
<td>0x01</td>
<td>0xFF</td>
<td>Sub type – End of Entire Device Path</td>
</tr>
<tr>
<td>0x22</td>
<td>0x02</td>
<td>0x04</td>
<td>Length – 0x04 bytes</td>
</tr>
</tbody>
</table>

15.7 Extended SCSI Pass Thru Protocol

This section defines the Extended SCSI Pass Thru Protocol. This protocol allows information about a SCSI channel to be collected, and allows SCSI Request Packets to be sent to any SCSI devices on a SCSI channel even if those devices are not boot devices. This protocol is attached to the device handle of each SCSI channel in a system that the protocol supports, and can be used for diagnostics. It may also be used to build a Block I/O driver for SCSI hard drives and SCSI CD-ROM or DVD drives to allow those devices to become boot devices. As ATAPI cmds are derived from SCSI cmds, the above statements also are applicable for ATAPI devices attached to a ATA controller. Packet-based commands(ATAPI cmds) would be sent to ATAPI devices only through the Extended SCSI Pass Thru Protocol.

EFI_EXT_SCSI_PASS_THRU_PROTOCOL

This section provides a detailed description of the EFI_EXT_SCSI_PASS_THRU_PROTOCOL.
Summary
Provides services that allow SCSI Pass Thru commands to be sent to SCSI devices attached to a SCSI channel. It also allows packet-based commands (ATAPI cmds) to be sent to ATAPI devices attached to a ATA controller.

GUID
#define EFI_EXT_SCSI_PASS_THRU_PROTOCOL_GUID \
{0x143b7632, 0xb81b, 0x4cb7,\} \
{0xab, 0xd3, 0xb6, 0x25, 0xa5, 0xb9, 0xbf, 0xfe}

Protocol Interface Structure
typedef struct _EFI_EXT_SCSI_PASS_THRU_PROTOCOL {
  EFI_EXT_SCSI_PASS_THRU_MODE   *Mode;
  EFI_EXT_SCSI_PASS_THRU_PASSTHRU PassThru;
  EFI_EXT_SCSI_PASS_THRU_GET_NEXT_TARGET_LUN GetNextTargetLun;
  EFI_EXT_SCSI_PASS_THRU_BUILD_DEVICE_PATH BuildDevicePath;
  EFI_EXT_SCSI_PASS_THRU_GET_TARGET_LUN GetTargetLun;
  EFI_EXT_SCSI_PASS_THRU_RESET_CHANNEL ResetChannel;
  EFI_EXT_SCSI_PASS_THRU_RESET_TARGET_LUN ResetTargetLun;
  EFI_EXT_SCSI_PASS_THRU_GET_NEXT_TARGET GetNextTarget;
} EFI_EXT_SCSI_PASS_THRU_PROTOCOL;

Parameters
Mode
A pointer to the EFI_EXT_SCSI_PASS_THRU_MODE data for this SCSI channel. EFI_EXT_SCSI_PASS_THRU_MODE is defined in “Related Definitions” below.

PassThru
Sends a SCSI Request Packet to a SCSI device that is Connected to the SCSI channel. See the PassThru() function description.

GetNextTargetLun
Retrieves the list of legal Target IDs and LUNs for the SCSI devices on a SCSI channel. See the GetNextTargetLun() function description.

BuildDevicePath
Allocates and builds a device path node for a SCSI Device on a SCSI channel. See the BuildDevicePath() function description.

GetTargetLun
Translates a device path node to a Target ID and LUN. See the GetTargetLun() function description.

ResetChannel
Resets the SCSI channel. This operation resets all the SCSI devices connected to the SCSI channel. See the ResetChannel() function description.

ResetTargetLun
Resets a SCSI device that is connected to the SCSI channel. See the ResetTargetLun() function description.

GetNextTarget
Retrieves the list of legal Target IDs for the SCSI devices on a SCSI channel. See the GetNextTarget() function description.

The following data values in the EFI_EXT_SCSI_PASS_THRU_MODE interface are read-only.

AdapterId
The Target ID of the host adapter on the SCSI channel.
Attributes

Additional information on the attributes of the SCSI channel. See “Related Definitions” below for the list of possible attributes.

IoAlign

Supplies the alignment requirement for any buffer used in a data transfer. IoAlign values of 0 and 1 mean that the buffer can be placed anywhere in memory. Otherwise, IoAlign must be a power of 2, and the requirement is that the start address of a buffer must be evenly divisible by IoAlign with no remainder.

Related Definitions

typedef struct {
    UINT32 AdapterId;
    UINT32 Attributes;
    UINT32 IoAlign;
} EFI_EXT_SCSI_PASS_THRU_MODE;

#define TARGET_MAX_BYTES 0x10
#define EFI_EXT_SCSI_PASS_THRU_ATTRIBUTES_PHYSICAL 0x0001
#define EFI_EXT_SCSI_PASS_THRU_ATTRIBUTES_LOGICAL 0x0002
#define EFI_EXT_SCSI_PASS_THRU_ATTRIBUTES_NONBLOCKIO 0x0004

EFI_EXT_SCSI_PASS_THRU_ATTRIBUTES_PHYSICAL

If this bit is set, then the EFI_EXT_SCSI_PASS_THRU_PROTOCOL interface is for physical devices on the SCSI channel.

EFI_EXT_SCSI_PASS_THRU_ATTRIBUTES_LOGICAL

If this bit is set, then the EFI_EXT_SCSI_PASS_THRU_PROTOCOL interface is for logical devices on the SCSI channel.

EFI_EXT_SCSI_PASS_THRU_ATTRIBUTES_NONBLOCKIO

If this bit is set, then the EFI_EXT_SCSI_PASS_THRU_PROTOCOL interface supports non blocking I/O. Every EFI_EXT_SCSI_PASS_THRU_PROTOCOL must support blocking I/O. The support of nonblocking I/O is optional.

Description

The EFI_EXT_SCSI_PASS_THRU_PROTOCOL provides information about a SCSI channel and the ability to send SCI Request Packets to any SCSI device attached to that SCSI channel. The information includes the Target ID of the host controller on the SCSI channel and the attributes of the SCSI channel.

The printable name for the SCSI controller, and the printable name of the SCSI channel can be provided through the EFI_COMPONENT_NAME2_PROTOCOL for multiple languages.

The Attributes field of the EFI_EXT_SCSI_PASS_THRU_PROTOCOL interface tells if the interface is for physical SCSI devices or logical SCSI devices. Drivers for non-RAID SCSI controllers will set both the EFI_EXT_SCSI_PASS_THRU_ATTRIBUTES_PHYSICAL and the EFI_EXT_SCSI_PASS_THRU_ATTRIBUTES_LOGICAL bits.

Drivers for RAID controllers that allow access to the physical devices and logical devices will produce two EFI_EXT_SCSI_PASS_THRU_PROTOCOL interfaces: one with the just the EFI_EXT_SCSI_PASS_THRU_ATTRIBUTES_PHYSICAL bit set and another with just the...
**EFI_EXT_SCSI_PASS_THRU_ATTRIBUTES_LOGICAL** bit set. One interface can be used to access the physical devices attached to the RAID controller, and the other can be used to access the logical devices attached to the RAID controller for its current configuration.

Drivers for RAID controllers that do not allow access to the physical devices will produce one **EFI_EXT_SCSI_PASS_THRU_ATTRIBUTES_PHYSICAL** bit set. The interface for logical devices can also be used by a file system driver to mount the RAID volumes. An **EFI_EXT_SCSI_PASS_THRU_PROTOCOL** with neither **EFI_EXT_SCSI_PASS_THRU_ATTRIBUTES_LOGICAL** nor **EFI_EXT_SCSI_PASS_THRU_ATTRIBUTES_PHYSICAL** set is an illegal configuration.

The Attributes field also contains the **EFI_EXT_SCSI_PASS_THRU_ATTRIBUTES_NONBLOCKIO** bit. All **EFI_EXT_SCSI_PASS_THRU_PROTOCOL** interfaces must support blocking I/O. If this bit is set, then the interface support both blocking I/O and nonblocking I/O.

Each **EFI_EXT_SCSI_PASS_THRU_PROTOCOL** instance must have an associated device path. Typically this will have an ACPI device path node and a PCI device path node, although variation will exist. For a SCSI controller that supports only one channel per PCI bus/device/function, it is recommended, but not required, that an additional Controller device path node (for controller 0) be appended to the device path.

For a SCSI controller that supports multiple channels per PCI bus/device/function, it is required that a Controller device path node be appended for each channel.

Additional information about the SCSI channel can be obtained from protocols attached to the same handle as the **EFI_EXT_SCSI_PASS_THRU_PROTOCOL**, or one of its parent handles. This would include the device I/O abstraction used to access the internal registers and functions of the SCSI controller.

**EFI_EXT_SCSI_PASS_THRU_PROTOCOL.PassThru()**

**Summary**

Sends a SCSI Request Packet to a SCSI device that is attached to the SCSI channel. This function supports both blocking I/O and nonblocking I/O. The blocking I/O functionality is required, and the nonblocking I/O functionality is optional.
Prototype

```c
typedef EFI_STATUS
(EFIAPI *EFI_EXT_SCSI_PASS_THRU_PASSTHRU) (  
    IN EFI_EXT_SCSI_PASS_THRU_PROTOCOL *This,  
    IN UINT8 *Target,  
    IN UINT64 Lun,  
    IN OUT EFI_EXT_SCSI_PASS_THRU_SCSI_REQUEST_PACKET *Packet,  
    IN EFI_EVENT Event OPTIONAL);
```

Parameters

- **This**: A pointer to the `EFI_EXT_SCSI_PASS_THRU_PROTOCOL` instance. Type `EFI_EXT_SCSI_PASS_THRU_PROTOCOL` is defined in Section 15.7.
- **Target**: The Target is an array of size `TARGET_MAX_BYTES` and it represents the id of the SCSI device to send the SCSI Request Packet. Each transport driver may chose to utilize a subset of this size to suit the needs of transport target representation. For example, a Fibre Channel driver may use only 8 bytes (WWN) to represent an FC target.
- **Lun**: The LUN of the SCSI device to send the SCSI Request Packet.
- **Packet**: A pointer to the SCSI Request Packet to send to the SCSI device specified by `Target` and `Lun`. See “Related Definitions” below for a description of `EFI_EXT_SCSI_PASS_THRU_SCSI_REQUEST_PACKET`.
- **Event**: If nonblocking I/O is not supported then `Event` is ignored, and blocking I/O is performed. If `Event` is `NULL`, then blocking I/O is performed. If `Event` is not `NULL` and non blocking I/O is supported, then nonblocking I/O is performed, and `Event` will be signaled when the SCSI Request Packet completes.
typedef struct {
    UINT64      Timeout;
    VOID*       InDataBuffer;
    VOID*       OutDataBuffer;
    VOID*       SenseData;
    VOID*       Cdb;
    UINT32      InTransferLength;
    UINT32      OutTransferLength;
    UINT8       CdbLength;
    UINT8       DataDirection;
    UINT8       HostAdapterStatus;
    UINT8       TargetStatus;
    UINT8       SenseDataLength;
} EFI_EXT_SCSI_PASS_THRU_SCSI_REQUEST_PACKET;

Timeout

The timeout, in 100 ns units, to use for the execution of this SCSI Request Packet. A Timeout value of 0 means that this function will wait indefinitely for the SCSI Request Packet to execute. If Timeout is greater than zero, then this function will return EFI_TIMEOUT if the time required to execute the SCSI Request Packet is greater than Timeout.

InDataBuffer

A pointer to the data buffer to transfer between the SCSI controller and the SCSI device for read and bidirectional commands. For all write and non data commands where InTransferLength is 0 this field is optional and may be NULL. If this field is not NULL, then it must be aligned on the boundary specified by the IoAlign field in the EFI_EXT_SCSI_PASS_THRU_MODE structure.

OutDataBuffer

A pointer to the data buffer to transfer between the SCSI controller and the SCSI device for write or bidirectional commands. For all read and non data commands where OutTransferLength is 0 this field is optional and may be NULL. If this field is not NULL, then it must be aligned on the boundary specified by the IoAlign field in the EFI_EXT_SCSI_PASS_THRU_MODE structure.

SenseData

A pointer to the sense data that was generated by the execution of the SCSI Request Packet. If SenseDataLength is 0, then this field is optional and may be NULL. It is strongly recommended that a sense data buffer of at least 252 bytes be provided to guarantee the entire sense data buffer generated from the execution of the SCSI Request Packet can be returned. If this field is not NULL, then it must be aligned to the boundary specified in the IoAlign field in the EFI_EXT_SCSI_PASS_THRU_MODE structure.

Cdb

A pointer to buffer that contains the Command Data Block to send to the SCSI device specified by Target and Lun.

InTransferLength

On input, the size, in bytes, of InDataBuffer. On output, the number of bytes transferred between the SCSI controller and the
SCSI device. If `InTransferLength` is larger than the SCSI controller can handle, no data will be transferred, `InTransferLength` will be updated to contain the number of bytes that the SCSI controller is able to transfer, and `EFI_BAD_BUFFER_SIZE` will be returned.

**OutTransferLength** On Input, the size, in bytes of `OutDataBuffer`. On Output, the number of bytes transferred between SCSI Controller and the SCSI device. If `OutTransferLength` is larger than the SCSI controller can handle, no data will be transferred, `OutTransferLength` will be updated to contain the number of bytes that the SCSI controller is able to transfer, and `EFI_BAD_BUFFER_SIZE` will be returned.

**CdbLength** The length, in bytes, of the buffer `Cdb`. The standard values are 6, 10, 12, and 16, but other values are possible if a variable length CDB is used.

**DataDirection** The direction of the data transfer. 0 for reads, 1 for writes. A value of 2 is Reserved for Bi-Directional SCSI commands. For example XDREADWRITE. All other values are reserved, and must not be used.

**HostAdapterStatus** The status of the host adapter specified by `This` when the SCSI Request Packet was executed on the target device. See the possible values listed below. If bit 7 of this field is set, then `HostAdapterStatus` is a vendor defined error code.

**TargetStatus** The status returned by the device specified by `Target` and `Lun` when the SCSI Request Packet was executed. See the possible values listed below.

**SenseDataLength** On input, the length in bytes of the `SenseData` buffer. On output, the number of bytes written to the `SenseData` buffer.
// DataDirection
//
#define EFI_EXT_SCSI_DATA_DIRECTION_READ           0
#define EFI_EXT_SCSI_DATA_DIRECTION_WRITE          1
#define EFI_EXT_SCSI_DATA_DIRECTION_BIDIRECTIONAL  2

// HostAdapterStatus
//
#define EFI_EXT_SCSI_STATUS_HOST_ADAPTER_OK                    0x00
#define EFI_EXT_SCSI_STATUS_HOST_ADAPTER_TIMEOUT_COMMAND       0x09
#define EFI_EXT_SCSI_STATUS_HOST_ADAPTER_TIMEOUT               0x0b
#define EFI_EXT_SCSI_STATUS_HOST_ADAPTER_MESSAGE_REJECT        0x0d
#define EFI_EXT_SCSI_STATUS_HOST_ADAPTER_BUS_RESET             0x0e
#define EFI_EXT_SCSI_STATUS_HOST_ADAPTER_PARITY_ERROR          0x0f
#define EFI_EXT_SCSI_STATUS_HOST_ADAPTER_REQUEST_SENSE_FAILED  0x10
#define EFI_EXT_SCSI_STATUS_HOST_ADAPTER_SELECTION_TIMEOUT     0x11
#define EFI_EXT_SCSI_STATUS_HOST_ADAPTER_DATA_OVERRUN_UNDERRUN 0x12
#define EFI_EXT_SCSI_STATUS_HOST_ADAPTER_BUS_FREE              0x13
#define EFI_EXT_SCSI_STATUS_HOST_ADAPTER_PHASE_ERROR           0x14
#define EFI_EXT_SCSI_STATUS_HOST_ADAPTER_OTHER                 0x7f

// TargetStatus
//
#define EFI_EXT_SCSI_STATUS_TARGET_GOOD                        0x00
#define EFI_EXT_SCSI_STATUS_TARGET_CHECK_CONDITION             0x02
#define EFI_EXT_SCSI_STATUS_TARGET_CONDITION_MET               0x04
#define EFI_EXT_SCSI_STATUS_TARGET_BUSY                        0x08
#define EFI_EXT_SCSI_STATUS_TARGET_INTERMEDIATE                0x10
#define EFI_EXT_SCSI_STATUS_TARGET_INTERMEDIATE_CONDITION_MET  0x14
#define EFI_EXT_SCSI_STATUS_TARGET_RESERVATION_CONFLICT        0x18
#define EFI_EXT_SCSI_STATUS_TARGET_TASK_SET_FULL               0x28
#define EFI_EXT_SCSI_STATUS_TARGET_ACA_ACTIVE                  0x30
#define EFI_EXT_SCSI_STATUS_TARGET_TASK_ABORTED                0x40

Description
The PassThru() function sends the SCSI Request Packet specified by Packet to the SCSI device specified by Target and Lun. If the driver supports nonblocking I/O and Event is not NULL, then the driver will return immediately after the command is sent to the selected device, and will later signal Event when the command has completed.

If the driver supports nonblocking I/O and Event is NULL, then the driver will send the command to the selected device and block until it is complete.

If the driver does not support nonblocking I/O, then the Event parameter is ignored, and the driver will send the command to the selected device and block until it is complete.
If **Packet** is successfully sent to the SCSI device, then EFI_SUCCESS is returned.

If **Packet** cannot be sent because there are too many packets already queued up, then EFI_NOT_READY is returned. The caller may retry **Packet** at a later time.

If a device error occurs while sending the **Packet**, then EFI_DEVICE_ERROR is returned.

If a timeout occurs during the execution of **Packet**, then EFI_TIMEOUT is returned.

If a device is not present but the target/LUN address in the packet are valid, then EFI_TIMEOUT is returned, and HostStatus is set to **EFI_EXT_SCSI_STATUS_HOST_ADAPTER_TIMEOUT_COMMAND**.

If **Target** or **Lun** are not in a valid range for the SCSI channel, then **EFI_INVALID_PARAMETER** is returned. If **InDataBuffer**, **OutDataBuffer** or **SenseData** do not meet the alignment requirement specified by the **IoAlign** field of the **EFI_EXT_SCSI_PASS_THRU_MODE** structure, then **EFI_INVALID_PARAMETER** is returned. If any of the other fields of **Packet** are invalid, then **EFI_INVALID_PARAMETER** is returned.

If the data buffer described by **InDataBuffer** and **InTransferLength** is too big to be transferred in a single command, then no data is transferred and **EFI_BAD_BUFFER_SIZE** is returned. The number of bytes that can be transferred in a single command are returned in **InTransferLength**.

If the data buffer described by **OutDataBuffer** and **OutTransferLength** is too big to be transferred in a single command, then no data is transferred and **EFI_BAD_BUFFER_SIZE** is returned. The number of bytes that can be transferred in a single command are returned in **OutTransferLength**.

If the command described in **Packet** is not supported by the host adapter, then **EFI_UNSUPPORTED** is returned.

If **EFI_SUCCESS**, **EFI_BAD_BUFFER_SIZE**, **EFI_DEVICE_ERROR**, or **EFI_TIMEOUT** is returned, then the caller must examine the status fields in **Packet** in the following precedence order: **HostAdapterStatus** followed by **TargetStatus** followed by **SenseDataLength**, followed by **SenseData**.

If nonblocking I/O is being used, then the status fields in **Packet** will not be valid until the **Event** associated with **Packet** is signaled.

If **EFI_NOT_READY**, **EFI_INVALID_PARAMETER** or **EFI_UNSUPPORTED** is returned, then **Packet** was never sent, so the status fields in **Packet** are not valid. If nonblocking I/O is being used, the **Event** associated with **Packet** will not be signaled.

Note: Some examples of SCSI read commands are READ, INQUIRY, and MODE_SENSE.

Note: Some examples of SCSI write commands are WRITE and MODE_SELECT.

Note: An example of a SCSI non data command is TEST_UNIT_READY.
## Status Codes Returned

<table>
<thead>
<tr>
<th>Code</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>EFI_SUCCESS</strong></td>
<td>The SCSI Request Packet was sent by the host. For bi-directional commands, \texttt{InTransferLength} bytes were transferred from \texttt{InDataBuffer}. For write and bi-directional commands, \texttt{OutTransferLength} bytes were transferred by \texttt{OutDataBuffer}. See \texttt{HostAdapterStatus}, \texttt{TargetStatus}, \texttt{SenseDataLength}, and \texttt{SenseData} in that order for additional status information.</td>
</tr>
<tr>
<td><strong>EFI_BAD_BUFFER_SIZE</strong></td>
<td>The SCSI Request Packet was not executed. The number of bytes that could be transferred is returned in \texttt{InTransferLength}. For write and bi-directional commands, \texttt{OutTransferLength} bytes were transferred by \texttt{OutDataBuffer}. See \texttt{HostAdapterStatus}, \texttt{TargetStatus}, and in that order for additional status information.</td>
</tr>
<tr>
<td><strong>EFI_NOT_READY</strong></td>
<td>The SCSI Request Packet could not be sent because there are too many SCSI Request Packets already queued. The caller may retry again later.</td>
</tr>
<tr>
<td><strong>EFI_DEVICE_ERROR</strong></td>
<td>A device error occurred while attempting to send the SCSI Request Packet. See \texttt{HostAdapterStatus}, \texttt{TargetStatus}, \texttt{SenseDataLength}, and \texttt{SenseData} in that order for additional status information.</td>
</tr>
<tr>
<td><strong>EFI_INVALID_PARAMETER</strong></td>
<td>Target, Lun, or the contents of \texttt{ScsiRequestPacket} are invalid. The SCSI Request Packet was not sent, so no additional status information is available.</td>
</tr>
<tr>
<td><strong>EFI_UNSUPPORTED</strong></td>
<td>The command described by the SCSI Request Packet is not supported by the host adapter. This includes the case of Bi-directional SCSI commands not supported by the implementation. The SCSI Request Packet was not sent, so no additional status information is available.</td>
</tr>
<tr>
<td><strong>EFI_TIMEOUT</strong></td>
<td>A timeout occurred while waiting for the SCSI Request Packet to execute. See \texttt{HostAdapterStatus}, \texttt{TargetStatus}, \texttt{SenseDataLength}, and \texttt{SenseData} in that order for additional status information.</td>
</tr>
</tbody>
</table>

**EFI_EXT_SCSI_PASS_THRU_PROTOCOL.GetNextTargetLun()**

**Summary**

Used to retrieve the list of legal Target IDs and LUNs for SCSI devices on a SCSI channel. These can either be the list SCSI devices that are actually present on the SCSI channel, or the list of legal Target Ids and LUNs for the SCSI channel. Regardless, the caller of this function must probe the Target ID and LUN returned to see if a SCSI device is actually present at that location on the SCSI channel.
Prototype

typedef

EFI_STATUS

(EIFIAPI *EFI_EXT_SCSI_PASS_THRU_GET_NEXT_TARGET_LUN) (   
    IN EFI_EXT_SCSI_PASS_THRU_PROTOCOL *This,
    IN OUT UINT8 **Target,
    IN OUT UINT64 *Lun

);

Parameters

This

A pointer to the EFI_EXT_SCSI_PASS_THRU_PROTOCOL instance. Type EFI_EXT_SCSI_PASS_THRU_PROTOCOL is defined in Section 15.7.

Target

On input, a pointer to a legal Target ID (an array of size TARGET_MAX_BYTES) for a SCSI device present on the SCSI channel. On output, a pointer to the next legal Target ID (an array of TARGET_MAX_BYTES) of a SCSI device on a SCSI channel. An input value of 0xFF’s (all bytes in the array are 0xFF) in the Target array retrieves the first legal Target ID for a SCSI device present on a SCSI channel.

Lun

On input, a pointer to the LUN of a SCSI device present on the SCSI channel. On output, a pointer to the LUN of the next SCSI device ID on a SCSI channel.

Description

The EFI_EXT_SCSI_PASS_THRU_PROTOCOL.GetNextTargetLun() function retrieves a list of legal Target ID and LUN of a SCSI channel. If on input a Target is specified by all 0xFF in the Target array, then the first legal Target ID and LUN for a SCSI device on a SCSI channel is returned in Target and Lun, and EFI_SUCCESS is returned.

If Target and Lun is a Target ID and LUN value that was returned on a previous call to GetNextTargetLun(), then the next legal Target ID and LUN for a SCSI device on the SCSI channel is returned in Target and Lun, and EFI_SUCCESS is returned.

If Target array is not all 0xFF’s and Target and Lun were not returned on a previous call to GetNextTargetLun(), then EFI_INVALID_PARAMETER is returned.

If Target and Lun are the Target ID and LUN of the last SCSI device on the SCSI channel, then EFI_NOT_FOUND is returned.

Status Codes Returned

<table>
<thead>
<tr>
<th>Status Code</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>EFI_SUCCESS</td>
<td>The Target ID and LUN of the next SCSI device on the SCSI channel was returned in Target and Lun.</td>
</tr>
<tr>
<td>EFI_NOT_FOUND</td>
<td>There are no more SCSI devices on this SCSI channel.</td>
</tr>
<tr>
<td>EFI_INVALID_PARAMETER</td>
<td>Target array is not all 0xFF’s, and Target and Lun were not returned on a previous call to GetNextTargetLun().</td>
</tr>
</tbody>
</table>
EFI_EXT_SCSI_PASS_THRU_PROTOCOL.BuildDevicePath()

Summary
Used to allocate and build a device path node for a SCSI device on a SCSI channel.

Prototype
```c
typedef EFI_STATUS (EFI_APPI *EFI_EXT_SCSI_PASS_THRU_BUILD_DEVICE_PATH) (
    IN EFI_EXT_SCSI_PASS_THRU_PROTOCOL *This,
    IN UINT8 *Target,
    IN UINT64 Lun
    OUT EFI_DEVICE_PATH_PROTOCOL **DevicePath
);
```

Parameters
- **This**: A pointer to the EFI_EXT_SCSI_PASS_THRU_PROTOCOL instance. Type EFI_EXT_SCSI_PASS_THRU_PROTOCOL is defined in Section 15.7.
- **Target**: The Target is an array of size TARGET_MAX_BYTES and it specifies the Target ID of the SCSI device for which a device path node is to be allocated and built. Transport drivers may chose to utilize a subset of this size to suit the representation of targets. For example, a Fibre Channel driver may use only 8 bytes (WWN) in the array to represent a FC target.
- **Lun**: The LUN of the SCSI device for which a device path node is to be allocated and built.
- **DevicePath**: A pointer to a single device path node that describes the SCSI device specified by Target and Lun. This function is responsible for allocating the buffer DevicePath with the boot service AllocatePool(). It is the caller's responsibility to free DevicePath when the caller is finished with DevicePath.

Description
The EFI_EXT_SCSI_PASS_THRU_PROTOCOL.BuildDevicePath() function allocates and builds a single device path node for the SCSI device specified by Target and Lun. If the SCSI device specified by Target and Lun are not present on the SCSI channel, then EFI_NOT_FOUND is returned. If DevicePath is NULL, then EFI_INVALID_PARAMETER is returned. If there are not enough resources to allocate the device path node, then EFI_OUT_OF_RESOURCES is returned. Otherwise, DevicePath is allocated with the boot service AllocatePool(), the contents of DevicePath are initialized to describe the SCSI device specified by Target and Lun, and EFI_SUCCESS is returned.
Status Codes Returned

<table>
<thead>
<tr>
<th>Code</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>EFI_SUCCESS</td>
<td>The device path node that describes the SCSI device specified by Target and Lun was allocated and returned in DevicePath.</td>
</tr>
<tr>
<td>EFI_NOT_FOUND</td>
<td>The SCSI devices specified by Target and Lun does not exist on the SCSI channel.</td>
</tr>
<tr>
<td>EFI_INVALID_PARAMETER</td>
<td>DevicePath is NULL.</td>
</tr>
<tr>
<td>EFI_OUT_OF_RESOURCES</td>
<td>There are not enough resources to allocate DevicePath.</td>
</tr>
</tbody>
</table>

EFI_EXT_SCSI_PASS_THRU_PROTOCOL.GetTargetLun()

Summary

Used to translate a device path node to a Target ID and LUN.

Prototype

typedef

EFI_STATUS

(EIFIAPI *EFI_EXT_SCSI_PASS_THRU_GET_TARGET_LUN) (
    IN EFI_EXT_SCSI_PASS_THRU_PROTOCOL *This,
    IN EFI_DEVICE_PATH_PROTOCOL *DevicePath
    OUT UINT8 **Target,
    OUT UINT64 *Lun
);

Parameters

This A pointer to the EFI_EXT_SCSI_PASS_THRU_PROTOCOL instance. Type EFI_EXT_SCSI_PASS_THRU_PROTOCOL is defined in Section 15.7.

DevicePath A pointer to the device path node that describes a SCSI device on the SCSI channel.

Target A pointer to the Target Array which represents the ID of a SCSI device on the SCSI channel.

Lun A pointer to the LUN of a SCSI device on the SCSI channel.

Description

The EFI_EXT_SCSI_PASS_THRU_PROTOCOL.GetTargetLun() function determines the Target ID and LUN associated with the SCSI device described by DevicePath. If DevicePath is a device path node type that the SCSI Pass Thru driver supports, then the SCSI Pass Thru driver will attempt to translate the contents DevicePath into a Target ID and LUN. If this translation is successful, then that Target ID and LUN are returned in Target and Lun, and EFI_SUCCESS is returned.

If DevicePath, Target, or Lun are NULL, then EFI_INVALID_PARAMETER is returned.

If DevicePath is not a device path node type that the SCSI Pass Thru driver supports, then EFI_UNSUPPORTED is returned.
If `DevicePath` is a device path node type that the SCSI Pass Thru driver supports, but there is not a valid translation from `DevicePath` to a Target ID and LUN, then `EFI_NOT_FOUND` is returned.

### Status Codes Returned

<table>
<thead>
<tr>
<th>Code</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>EFI_SUCCESS</td>
<td><code>DevicePath</code> was successfully translated to a Target ID and LUN, and they were returned in Target and Lun.</td>
</tr>
<tr>
<td>EFI_INVALID_PARAMETER</td>
<td><code>DevicePath</code> is <code>NULL</code>.</td>
</tr>
<tr>
<td>EFI_INVALID_PARAMETER</td>
<td><code>Target</code> is <code>NULL</code>.</td>
</tr>
<tr>
<td>EFI_INVALID_PARAMETER</td>
<td><code>Lun</code> is <code>NULL</code>.</td>
</tr>
<tr>
<td>EFI_UNSUPPORTED</td>
<td>This driver does not support the device path node type in <code>DevicePath</code>.</td>
</tr>
<tr>
<td>EFI_NOT_FOUND</td>
<td>A valid translation from <code>DevicePath</code> to a Target ID and LUN does not exist.</td>
</tr>
</tbody>
</table>

### EFI_EXT_SCSI_PASS_THRU_PROTOCOL.ResetChannel()

**Summary**

Resets a SCSI channel. This operation resets all the SCSI devices connected to the SCSI channel.

**Prototype**

```c
typedef EFI_STATUS
  (EFIAPI *EFI_EXT_SCSI_PASS_THRU_RESET_CHANNEL) (IN EFI_EXT_SCSI_PASS_THRU_PROTOCOL *This);
```

**Parameters**

- `This` A pointer to the `EFI_EXT_SCSI_PASS_THRU_PROTOCOL` instance. Type `EFI_EXT_SCSI_PASS_THRU_PROTOCOL` is defined in Section 15.7.

**Description**

The `EFI_EXT_SCSI_PASS_THRU_PROTOCOL.ResetChannel()` function resets a SCSI channel. This operation resets all the SCSI devices connected to the SCSI channel. If this SCSI channel does not support a reset operation, then `EFI_UNSUPPORTED` is returned.

If a device error occurs while executing that channel reset operation, then `EFI_DEVICE_ERROR` is returned.

If a timeout occurs during the execution of the channel reset operation, then `EFI_TIMEOUT` is returned.

If the channel reset operation is completed, then `EFI_SUCCESS` is returned.
Status Codes Returned

<table>
<thead>
<tr>
<th>Code</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>EFI_SUCCESS</td>
<td>The SCSI channel was reset.</td>
</tr>
<tr>
<td>EFI_UNSUPPORTED</td>
<td>The SCSI channel does not support a channel reset operation.</td>
</tr>
<tr>
<td>EFI_DEVICE_ERROR</td>
<td>A device error occurred while attempting to reset the SCSI channel.</td>
</tr>
<tr>
<td>EFI_TIMEOUT</td>
<td>A timeout occurred while attempting to reset the SCSI channel.</td>
</tr>
</tbody>
</table>

EFI_EXT_SCSI_PASS_THRU_PROTOCOL.ResetTargetLun()

Summary
Resets a SCSI logical unit that is connected to a SCSI channel.

Prototype
```c
typedef EFI_STATUS
  (EFIAPI *EFI_EXT_SCSI_PASS_THRU_RESET_TARGET_LUN) (
   IN EFI_EXT_SCSI_PASS_THRU_PROTOCOL  *This,
   IN UINT8                        *Target,
   IN UINT64                      Lun
  );
```

Parameters
- **This**: A pointer to the EFI_EXT_SCSI_PASS_THRU_PROTOCOL instance. Type EFI_EXT_SCSI_PASS_THRU_PROTOCOL is defined in Section 15.7.
- **Target**: The Target is an array of size TARGET_MAX_BYTE and it represents the target port ID of the SCSI device containing the SCSI logical unit to reset. Transport drivers may chose to utilize a subset of this array to suit the representation of their targets. For example a Fibre Channel driver may use only 8 bytes in the array (WWN) to represent a FC target.
- **Lun**: The LUN of the SCSI device to reset.

Description
The EFI_EXT_SCSI_PASS_THRU_PROTOCOL.ResetTargetLun() function resets the SCSI logical unit specified by Target and Lun. If this SCSI channel does not support a target reset operation, then EFI_UNSUPPORTED is returned.

If Target or Lun are not in a valid range for this SCSI channel, then EFI_INVALID_PARAMETER is returned.

If a device error occurs while executing that logical unit reset operation, then EFI_DEVICE_ERROR is returned.

If a timeout occurs during the execution of the logical unit reset operation, then EFI_TIMEOUT is returned.

If the logical unit reset operation is completed, then EFI_SUCCESS is returned.
Status Codes Returned

<table>
<thead>
<tr>
<th>Code</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>EFI_SUCCESS</td>
<td>The SCSI device specified by Target and Lun was reset</td>
</tr>
<tr>
<td>EFI_UNSUPPORTED</td>
<td>The SCSI channel does not support a target reset operation.</td>
</tr>
<tr>
<td>EFI_INVALID_PARAMETER</td>
<td>Target or Lun are invalid.</td>
</tr>
<tr>
<td>EFI_DEVICE_ERROR</td>
<td>A device error occurred while attempting to reset the SCSI device specified by Target and Lun.</td>
</tr>
<tr>
<td>EFI_TIME_OUT</td>
<td>A timeout occurred while attempting to reset the SCSI device specified by Target and Lun.</td>
</tr>
</tbody>
</table>

**EFI_EXT_SCSI_PASS_THRU_PROTOCOL.GetNextTarget()**

**Summary**

Used to retrieve the list of legal Target IDs for SCSI devices on a SCSI channel. These can either be the list SCSI devices that are actually present on the SCSI channel, or the list of legal Target IDs for the SCSI channel. Regardless, the caller of this function must probe the Target ID returned to see if a SCSI device is actually present at that location on the SCSI channel.

**Prototype**

```c
typedef EFI_STATUS
  (EFIAPI *EFI_EXT_SCSI_PASS_THRU_GET_NEXT_TARGET) (    
    IN EFI_EXT_SCSI_PASS_THRU_PROTOCOL    *This,    
    IN OUT UINT8                         **Target,    
  );
```

**Parameters**

- **This**
  A pointer to the `EFI_EXT_SCSI_PASS_THRU_PROTOCOL` instance. Type `EFI_EXT_SCSI_PASS_THRU_PROTOCOL` is defined in Section 15.7.

- **Target**
  On input, a pointer to the Target ID (an array of size `TARGET_MAX_BYTES`) of a SCSI device present on the SCSI channel. On output, a pointer to the Target ID (an array of `TARGET_MAXgetBytes`) of the next SCSI device present on a SCSI channel. An input value of `0xFF`'s (all bytes in the array are `0xFF`) in the Target array retrieves the Target ID of the first SCSI device present on a SCSI channel.

**Description**

The `EFI_EXT_SCSI_PASS_THRU_PROTOCOL.GetNextTarget()` function retrieves the Target ID of a SCSI device present on a SCSI channel. If on input a Target is specified by all `0xF` in the Target array, then the Target ID of the first SCSI device is returned in Target and `EFI_SUCCESS` is returned.

If Target is a Target ID value that was returned on a previous call to `GetNextTarget()`, then the Target ID of the next SCSI device on the SCSI channel is returned in Target, and `EFI_SUCCESS` is returned.
If Target array is not all 0xFF’s and Target were not returned on a previous call to GetNextTarget(), then EFI_INVALID_PARAMETER is returned.

If Target is the Target ID of the last SCSI device on the SCSI channel, then EFI_NOT_FOUND is returned.

**Status Codes Returned**

<table>
<thead>
<tr>
<th>Code</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>EFI_SUCCESS</td>
<td>The Target ID of the next SCSI device on the SCSI channel was returned in Target.</td>
</tr>
<tr>
<td>EFI_NOT_FOUND</td>
<td>There are no more SCSI devices on this SCSI channel.</td>
</tr>
<tr>
<td>EFI_INVALID_PARAMETER</td>
<td>Target array is not all 0xFF’s, and Target were not returned on a previous call to GetNextTarget().</td>
</tr>
</tbody>
</table>
16 - Protocols — iSCSI Boot

16.1 Overview

The iSCSI protocol defines a transport for SCSI data over TCP/IP. It also provides an interoperable solution that takes advantage of existing internet infrastructure, management facilities, and addresses distance limitations. The iSCSI protocol specification was developed by the Internet Engineering Task Force (IETF) and is SCSI Architecture Model-2 (SAM-2) compliant. iSCSI encapsulates block-oriented SCSI commands into iSCSI Protocol Data Units (PDU) that traverse the network over TCP/IP. iSCSI defines a Session, the initiator and target nexus (I-T nexus), which could be a bundle of one or more TCP connections.

Similar to other existing mass storage protocols like Fibre Channel and parallel SCSI, boot over iSCSI is an important functionality. This document will attempt to capture the various cases for iSCSI boot and common up with generic EFI protocol changes to address them.

16.1.1 iSCSI UEFI Driver Layering

iSCSI UEFI Drivers may exist in two different forms:

- iSCSI UEFI Driver on a NIC:
  
  The driver will be layered on top of the networking layers. It will use the DHCP, IP, and TCP and packet level interface protocols of the UEFI networking stack. The driver will use an iSCSI software initiator.

- iSCSI UEFI Driver on a Host Bus Adapter (HBA) that may use an offloading engine such as TOE (or any other TCP offload card):
  
  The driver will be layered on top of the TOE TCP interfaces. It will use the DHCP, IP, TCP protocols of the TOE. The driver will present itself as a SCSI device driver using interfaces such as `EFI_EXT_SCSI_PASS_THRU_PROTOCOL`.

To help in detecting iSCSI UEFI Drivers and their capabilities, the iSCSI UEFI driver handle must include an instance of the `EFI_ADAPTER_INFORMATION_PROTOCOL` with a `EFI_ADAPTER_INFO_NETWORK_BOOT` structure.

16.2 EFI iSCSI Initiator Name Protocol

This protocol sets and obtains the iSCSI Initiator Name. The iSCSI Initiator Name protocol builds a default iSCSI name. The iSCSI name configures using the programming interfaces defined below. Successive configuration of the iSCSI initiator name overwrites the previously existing name. Once overwritten, the previous name will not be retrievable. Setting an iSCSI name string that is zero length is illegal. The maximum size of the iSCSI Initiator Name is 224 bytes (including the NULL terminator).

**EFI_ISCSI_INITIATOR_NAME_PROTOCOL**

**Summary**

iSCSI Initiator Name Protocol for setting and obtaining the iSCSI Initiator Name.
GUID

```c
#define EFI_ISCSI_INITIATOR_NAME_PROTOCOL_GUID \
  {0x59324945, 0xec44, 0x4c0d, \ 
    {0xb1, 0xcd, 0x9d, 0xb1, 0x39, 0xdf, 0x07, 0x0c}}
```

Protocol Interface Structure

```c
typedef struct _EFI_ISCSI_INITIATOR_NAME_PROTOCOL {
  EFI_ISCSI_INITIATOR_NAME_GET Get;
  EFI_ISCSI_INITIATOR_NAME_SET Set;
} EFI_ISCSI_INITIATOR_NAME_PROTOCOL;
```

Parameters

- **Get**: Used to retrieve the iSCSI Initiator Name.
- **Set**: Used to set the iSCSI Initiator Name.

Description

The `EFI_ISCSI_INITIATOR_NAME_PROTOCOL` provides the ability to get and set the iSCSI Initiator Name.

**EFI_ISCSI_INITIATOR_NAME_PROTOCOL. Get()**

Summary

Retrieves the current set value of iSCSI Initiator Name.

Prototype

```c
typedef EFI_STATUS
  (EFIAPI *EFI_ISCSI_INITIATOR_NAME_GET) (\n   IN   EFI_ISCSI_INITIATOR_NAME_PROTOCOL *This, \n   IN OUT UINTN *BufferSize, \n   OUT   VOID *Buffer \n  );
```

Parameters

- **This**: Pointer to the `EFI_ISCSI_INITIATOR_NAME_PROTOCOL` instance.
- **BufferSize**: Size of the buffer in bytes pointed to by Buffer / Actual size of the variable data buffer.
- **Buffer**: Pointer to the buffer for data to be read. The data is a null-terminated UTF-8 encoded string. The maximum length is 223 characters, including the null-terminator.

Description

This function will retrieve the iSCSI Initiator Name from Non-volatile memory.
Status Codes Returned

<table>
<thead>
<tr>
<th>Code</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>EFI_SUCCESS</td>
<td>Data was successfully retrieved into the provided buffer and the BufferSize was sufficient to handle the iSCSI initiator name</td>
</tr>
<tr>
<td>EFI_BUFFER_TOO_SMALL</td>
<td>BufferSize is too small for the result. BufferSize will be updated with the size required to complete the request. Buffer will not be affected.</td>
</tr>
<tr>
<td>EFI_INVALID_PARAMETER</td>
<td>BufferSize is NULL. BufferSize and Buffer will not be affected.</td>
</tr>
<tr>
<td>EFI_INVALID_PARAMETER</td>
<td>Buffer is NULL. BufferSize and Buffer will not be affected.</td>
</tr>
<tr>
<td>EFI_DEVICE_ERROR</td>
<td>The iSCSI initiator name could not be retrieved due to a hardware error.</td>
</tr>
</tbody>
</table>

**EFI_ISCSI_INITIATOR_NAME_PROTOCOL.Set()**

**Summary**
Sets the iSCSI Initiator Name.

**Prototype**
```
typedef EFI_STATUS
(EFIAPI *EFI_ISCSI_INITIATOR_NAME_SET) (  
    IN   EFI_ISCSI_INITIATOR_NAME_PROTOCOL *This,
    IN OUT UINTN *BufferSize,
    IN   VOID *Buffer
);
```

**Parameters**
- **This** Pointer to the **EFI_ISCSI_INITIATOR_NAME_PROTOCOL** instance
- **BufferSize** Size of the buffer in bytes pointed to by Buffer.
- **Buffer** Pointer to the buffer for data to be written. The data is a null-terminated UTF-8 encoded string. The maximum length is 223 characters, including the null-terminator.

**Description**
This function will set the iSCSI Initiator Name into Non-volatile memory.
## Status Codes Returned

<table>
<thead>
<tr>
<th>Status Code</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>EFI_SUCCESS</td>
<td>Data was successfully stored by the protocol</td>
</tr>
<tr>
<td>EFI_UNSUPPORTED</td>
<td>Platform policies do not allow for data to be written</td>
</tr>
<tr>
<td>EFI_INVALID_PARAMETER</td>
<td>BufferSize exceeds the maximum allowed limit. BufferSize will be updated with the maximum size required to complete the request.</td>
</tr>
<tr>
<td>EFI_INVALID_PARAMETER</td>
<td>Buffersize is NULL. BufferSize and Buffer will not be affected</td>
</tr>
<tr>
<td>EFI_INVALID_PARAMETER</td>
<td>Buffer is NULL. BufferSize and Buffer will not be affected.</td>
</tr>
<tr>
<td>EFI_DEVICE_ERROR</td>
<td>The data could not be stored due to a hardware error.</td>
</tr>
<tr>
<td>EFI_OUT_OF_RESOURCES</td>
<td>Not enough storage is available to hold the data</td>
</tr>
<tr>
<td>EFI_PROTOCOL_ERROR</td>
<td>Input iSCSI initiator name does not adhere to RFC 3720 (and other related protocols)</td>
</tr>
</tbody>
</table>
17 - Protocols — USB Support

17.1 USB2 Host Controller Protocol

Section 17.1 and Section 17.1.1 describe the USB2 Host Controller Protocol. This protocol provides an I/O abstraction for a USB2 Host Controller. The USB2 Host Controller is a hardware component that interfaces to a Universal Serial Bus (USB). It moves data between system memory and devices on the USB by processing data structures and generating transactions on the USB. This protocol is used by a USB Bus Driver to perform all data transaction over the Universal Serial Bus. It also provides services to manage the USB root hub that is integrated into the USB Host Controller. USB device drivers do not use this protocol directly. Instead, they use the I/O abstraction produced by the USB Bus Driver. This protocol should only be used by drivers that require direct access to the USB bus.

17.1.1 USB Host Controller Protocol Overview

The USB Host Controller Protocol is used by code, typically USB bus drivers, running in the EFI boot services environment, to perform data transactions over a USB bus. In addition, it provides an abstraction for the root hub of the USB bus.

The interfaces provided in the EFI_USB2_HC_PROTOCOL are used to manage data transactions on a USB bus. It also provides control methods for the USB root hub. The EFI_USB2_HC_PROTOCOL is designed to support both USB 1.1 and USB 2.0 – compliant host controllers.

The EFI_USB2_HC_PROTOCOL abstracts basic functionality that is designed to operate with the EHCI, UHCI and OHCI standards. By using this protocol, a single USB bus driver can be implemented without knowing if the underlying USB host controller conforms to the XHCI, EHCI, OHCI or the UHCI standards.

Each instance of the EFI_USB2_HC_PROTOCOL corresponds to a USB host controller in a platform. The protocol is attached to the device handle of a USB host controller that is created by a device driver for the USB host controller’s parent bus type. For example, a USB host controller that is implemented as a PCI device would require a PCI device driver to produce an instance of the EFI_USB2_HC_PROTOCOL.

EFI_USB2_HC_PROTOCOL

Summary

Provides basic USB host controller management, basic data transactions over USB bus, and USB root hub access.

GUID

```plaintext
#define EFI_USB2_HC_PROTOCOL_GUID "0x3e745226,0x9818,0x45b6,0xa2,0xac,0xd7,0xcd,0xe,0x8b,0xa2,0xbc"
```

Protocol Interface Structure

```plaintext
typedef struct _EFI_USB2_HC_PROTOCOL {
    EFI_USB2_HC_PROTOCOL_GET_CAPABILITY GetCapability;
    EFI_USB2_HC_PROTOCOL_RESET Reset;
    EFI_USB2_HC_PROTOCOL_GET_STATE GetState;
} EFI_USB2_HC_PROTOCOL;
```
EFI_USB2_HC_PROTOCOL_SET_STATE  SetState;
EFI_USB2_HC_PROTOCOL_CONTROL_TRANSFER  ControlTransfer;
EFI_USB2_HC_PROTOCOL_BULK_TRANSFER  BulkTransfer;
EFI_USB2_HC_PROTOCOL_ASYNC_INTERRUPT_TRANSFER  AsyncInterruptTransfer;
EFI_USB2_HC_PROTOCOL_SYNC_INTERRUPT_TRANSFER  SyncInterruptTransfer;
EFI_USB2_HC_PROTOCOL_ISOCHRONOUS_TRANSFER  IsochronousTransfer;
EFI_USB2_HC_PROTOCOL_ASYNC_ISOCHRONOUS_TRANSFER  AsyncIsochronousTransfer;
EFI_USB2_HC_PROTOCOL_GET_ROOTHUB_PORT_STATUS  GetRootHubPortStatus;
EFI_USB2_HC_PROTOCOL_SET_ROOTHUB_PORT_FEATURE  SetRootHubPortFeature;
EFI_USB2_HC_PROTOCOL_CLEAR_ROOTHUB_PORT_FEATURE  ClearRootHubPortFeature;
UINT16  MajorRevision;
UINT16  MinorRevision;
} EFI_USB2_HC_PROTOCOL;

Parameters

GetCapability  Retrieves the capabilities of the USB host controller. See the GetCapability() function description.
Reset  Software reset of USB. See the Reset() function description.
GetState  Retrieves the current state of the USB host controller. See the GetState() function description.
SetState  Sets the USB host controller to a specific state. See the SetState() function description.
ControlTransfer  Submits a control transfer to a target USB device. See the ControlTransfer() function description.
BulkTransfer  Submits a bulk transfer to a bulk endpoint of a USB device. See the BulkTransfer() function description.
AsyncInterruptTransfer  Submits an asynchronous interrupt transfer to an interrupt endpoint of a USB device. See the AsyncInterruptTransfer() function description.
SyncInterruptTransfer  Submits a synchronous interrupt transfer to an interrupt endpoint of a USB device. See the SyncInterruptTransfer() function description.
IsochronousTransfer  Submits isochronous transfer to an isochronous endpoint of a USB device. See the IsochronousTransfer() function description.
AsyncIsochronousTransfer  Submits nonblocking USB isochronous transfer. See the AsyncIsochronousTransfer() function description.
GetRootHubPortStatus
Retrieves the status of the specified root hub port. See the
GetRootHubPortStatus() function description.

SetRootHubPortFeature
Sets the feature for the specified root hub port. See the
SetRootHubPortFeature() function description.

ClearRootHubPortFeature
Clears the feature for the specified root hub port. See the
ClearRootHubPortFeature() function description.

MajorRevision
The major revision number of the USB host controller. The revision
information indicates the release of the Universal Serial Bus
Specification with which the host controller is compliant.

MinorRevision
The minor revision number of the USB host controller. The revision
information indicates the release of the Universal Serial Bus
Specification with which the host controller is compliant.

Description
The EFI_USB2_HC_PROTOCOL provides USB host controller management, basic data transactions over a
USB bus, and USB root hub access. A device driver that wishes to manage a USB bus in a system retrieves
the EFI_USB2_HC_PROTOCOL instance that is associated with the USB bus to be managed. A device
handle for a USB host controller will minimally contain an EFI_DEVICE_PATH_PROTOCOL instance, and
an EFI_USB2_HC_PROTOCOL instance.

EFI_USB2_HC_PROTOCOL.GetCapability()

Summary
Retrieves the Host Controller capabilities.

Prototype
typedef
EFI_STATUS
(EIFIAPI *EFI_USB2_HC_PROTOCOL_GET_CAPABILITY) (  
  IN EFI_USB2_HC_PROTOCOL *This,
  OUT UINT8 *MaxSpeed,
  OUT UINT8 *PortNumber,
  OUT UINT8 *Is64BitCapable
);

Parameters
This A pointer to the EFI_USB2_HC_PROTOCOL instance. Type
EFI_USB2_HC_PROTOCOL is defined in Section 17.1.
MaxSpeed Host controller data transfer speed; see “Related Definitions” below
for a list of supported transfer speed values.
PortNumber Number of the root hub ports.
Is64BitCapable TRUE if controller supports 64-bit memory addressing, FALSE
otherwise.
Related Definitions

```
#define EFI_USB_SPEED_FULL  0x0000
#define EFI_USB_SPEED_LOW   0x0001
#define EFI_USB_SPEED_HIGH  0x0002
#define EFI_USB_SPEED_SUPER 0x0003
```

<table>
<thead>
<tr>
<th>Definition</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>EFI_USB_SPEED_LOW</td>
<td>Low speed USB device; data bandwidth is up to 1.5 Mb/s. Supported by USB 1.1 OHCI and UHCI host controllers.</td>
</tr>
<tr>
<td>EFI_USB_SPEED_FULL</td>
<td>Full speed USB device; data bandwidth is up to 12 Mb/s. Supported by USB 1.1 OHCI and UHCI host controllers.</td>
</tr>
<tr>
<td>EFI_USB_SPEED_HIGH</td>
<td>High speed USB device; data bandwidth is up to 480 Mb/s. Supported by USB 2.0 EHCI host controllers.</td>
</tr>
<tr>
<td>EFI_USB_SPEED_SUPER</td>
<td>Super speed USB device; data bandwidth is up to 4.8Gbs. Supported by USB 3.0 XHCI host controllers.</td>
</tr>
</tbody>
</table>

Description

This function is used to retrieve the host controller capabilities. **MaxSpeed** indicates the maximum data transfer speed the controller is capable of; this information is needed for the subsequent transfers. **PortNumber** is the number of root hub ports, it is required by the USB bus driver to perform bus enumeration. **Is64BitCapable** indicates that controller is capable of 64-bit memory access so that the host controller software can use memory blocks above 4 GiB for the data transfers.

Status Codes Returned

<table>
<thead>
<tr>
<th>Code</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>EFI_SUCCESS</td>
<td>The host controller capabilities were retrieved successfully.</td>
</tr>
<tr>
<td>EFI_INVALID_PARAMETER</td>
<td>MaxSpeed or PortNumber or Is64BitCapable is <strong>NULL</strong>.</td>
</tr>
<tr>
<td>EFI_DEVICE_ERROR</td>
<td>An error was encountered while attempting to retrieve the capabilities.</td>
</tr>
</tbody>
</table>

**EFI_USB2_HC_PROTOCOL.Reset()**

Summary

Provides software reset for the USB host controller.

Prototype

```
typedef
EFI_STATUS
(EFI_API *EFI_USB2_HC_PROTOCOL_RESET)(
    IN EFI_USB2_HC_PROTOCOL *This,
    IN UINT16 Attributes
);
```

Parameters

- **This** A pointer to the EFI_USB2_HC_PROTOCOL instance. Type **EFI_USB2_HC_PROTOCOL** is defined in Section 17.1.
- **Attributes** A bit mask of the reset operation to perform. See “Related Definitions” below for a list of the supported bit mask values.
Related Definitions

```c
#define EFI_USB_HC_RESET_GLOBAL    0x0001
#define EFI_USB_HC_RESET_HOST_CONTROLLER 0x0002
#define EFI_USB_HC_RESET_GLOBAL_WITH_DEBUG 0x0004
#define EFI_USB_HC_RESET_HOST_WITH_DEBUG   0x0008
```

**EFI_USB_HC_RESET_GLOBAL**

If this bit is set, a global reset signal will be sent to the USB bus. This resets all of the USB bus logic, including the USB host controller hardware and all the devices attached on the USB bus.

**EFI_USB_HC_RESET_HOST_CONTROLLER**

If this bit is set, the USB host controller hardware will be reset. No reset signal will be sent to the USB bus.

**EFI_USB_HC_RESET_GLOBAL_WITH_DEBUG**

If this bit is set, then a global reset signal will be sent to the USB bus. This resets all of the USB bus logic, including the USB host controller and all of the devices attached on the USB bus. If this is an XHCI or EHCI controller and the debug port has been configured, then this will still reset the host controller.

**EFI_USB_HC_RESET_HOST_WITH_DEBUG**

If this bit is set, the USB host controller hardware will be reset. If this is an XHCI or EHCI controller and the debug port has been configured, then this will still reset the host controller.

**Description**

This function provides a software mechanism to reset a USB host controller. The type of reset is specified by the `Attributes` parameter. If the type of reset specified by `Attributes` is not valid, then `EFI_INVALID_PARAMETER` is returned. If the reset operation is completed, then `EFI_SUCCESS` is returned. If the type of reset specified by `Attributes` is not currently supported by the host controller hardware, `EFI_UNSUPPORTED` is returned. If a device error occurs during the reset operation, then `EFI_DEVICE_ERROR` is returned.

Note: For XHCI or EHCI controllers, the `EFI_USB_HC_RESET_GLOBAL` and `EFI_USB_HC_RESET_HOST_CONTROLLER` types of reset do not actually reset the bus if the debug port has been configured. In these cases, the function will return `EFI_ACCESS_DENIED`. 
Status Codes Returned

<table>
<thead>
<tr>
<th>Code</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>EFI_SUCCESS</td>
<td>The reset operation succeeded.</td>
</tr>
<tr>
<td>EFI_INVALID_PARAMETER</td>
<td>Attributes is not valid.</td>
</tr>
<tr>
<td>EFI_UNSUPPORTED</td>
<td>The type of reset specified by Attributes is not currently supported by the host controller hardware.</td>
</tr>
<tr>
<td>EFI_ACCESS_DENIED</td>
<td>Reset operation is rejected due to the debug port being configured and active; only EFI_USB_HC_RESET_GLOBAL_WITH_DEBUG or EFI_USB_HC_RESET_HOST_WITH_DEBUG reset Attributes can be used to perform reset operation for this host controller.</td>
</tr>
<tr>
<td>EFI_DEVICE_ERROR</td>
<td>An error was encountered while attempting to perform the reset operation.</td>
</tr>
</tbody>
</table>

**EFI_USB2_HC_PROTOCOL.GetState()**

**Summary**
Retrieves current state of the USB host controller.

**Prototype**
```c
typedef
  EFI_STATUS
  (EFIAPI *EFI_USB2_HC_PROTOCOL_GET_STATE) (  
    IN EFI_USB2_HC_PROTOCOL  *This,  
    OUT EFI_USB_HC_STATE  *State );
```

**Parameters**
- **This**
  A pointer to the EFI_USB2_HC_PROTOCOL instance. Type EFI_USB2_HC_PROTOCOL is defined in Section 17.1.
- **State**
  A pointer to the EFI_USB_HC_STATE data structure that indicates current state of the USB host controller. Type EFI_USB_HC_STATE is defined in “Related Definitions.”

**Related Definitions**
```c
typedef enum {  
  EfiUsbHcStateHalt,  
  EfiUsbHcStateOperational,  
  EfiUsbHcStateSuspend,  
  EfiUsbHcStateMaximum  
  } EFI_USB_HC_STATE;
```

**EfiUsbHcStateHalt**
The host controller is in halt state. No USB transactions can occur while in this state. The host controller can enter this state for three reasons:
- After host controller hardware reset.
- Explicitly set by software.
• Triggered by a fatal error such as consistency check failure.

**EfiUsbHcStateOperational**

The host controller is in an operational state. When in this state, the host controller can execute bus traffic. This state must be explicitly set to enable the USB bus traffic.

**EfiUsbHcStateSuspend**

The host controller is in the suspend state. No USB transactions can occur while in this state. The host controller enters this state for the following reasons:

• Explicitly set by software.
• Triggered when there is no bus traffic for 3 microseconds.

**Description**

This function is used to retrieve the USB host controller’s current state. The USB Host Controller Protocol publishes three states for USB host controller, as defined in “Related Definitions” below. If **State** is **NULL**, then **EFI_INVALID_PARAMETER** is returned. If a device error occurs while attempting to retrieve the USB host controllers current state, then **EFI_DEVICE_ERROR** is returned. Otherwise, the USB host controller’s current state is returned in **State**, and **EFI_SUCCESS** is returned.

**Status Codes Returned**

<table>
<thead>
<tr>
<th>Status Code</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>EFI_SUCCESS</td>
<td>The state information of the host controller was returned in <strong>State</strong>.</td>
</tr>
<tr>
<td>EFI_INVALID_PARAMETER</td>
<td>State is <strong>NULL</strong>.</td>
</tr>
<tr>
<td>EFI_DEVICE_ERROR</td>
<td>An error was encountered while attempting to retrieve the host controller’s current state.</td>
</tr>
</tbody>
</table>

**EFI_USB2_HC_PROTOCOL.SetState()**

**Summary**

Sets the USB host controller to a specific state.

**Prototype**

```c
typedef
EFI_STATUS
(EIFIAPI *EFI_USB2_HC_PROTOCOL_SET_STATE) (  
    IN EFI_USB2_HC_PROTOCOL  *This,
    IN EFI_USB_HC_STATE  State
);
```

**Parameters**

- **This**
  A pointer to the EFI_USB2_HC_PROTOCOL instance. Type EFI_USB2_HC_PROTOCOL is defined in **Section 17.1**.
- **State**
  Indicates the state of the host controller that will be set. See the definition and description of the type EFI_USB_HC_STATE in the **GetState()** function description.
Description

This function is used to explicitly set a USB host controller’s state. There are three states defined for the USB host controller. These are the halt state, the operational state and the suspend state. Figure 17-1 illustrates the possible state transitions:

![Figure 17-1 Software Triggered State Transitions of a USB Host Controller](image)

If the state specified by State is not valid, then EFI_INVALID_PARAMETER is returned. If a device error occurs while attempting to place the USB host controller into the state specified by State, then EFI_DEVICE_ERROR is returned. If the USB host controller is successfully placed in the state specified by State, then EFI_SUCCESS is returned.

**Status Codes Returned**

<table>
<thead>
<tr>
<th>Status Code</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>EFI_SUCCESS</td>
<td>The USB host controller was successfully placed in the state specified by State.</td>
</tr>
<tr>
<td>EFI_INVALID_PARAMETER</td>
<td>State is invalid.</td>
</tr>
<tr>
<td>EFI_DEVICE_ERROR</td>
<td>Failed to set the state specified by State due to device error.</td>
</tr>
</tbody>
</table>

**EFI_USB2_HC_PROTOCOL.ControlTransfer()**

**Summary**

Submits control transfer to a target USB device.
Prototype

typedef
  EFI_STATUS
  (EFIAPI *EFI_USB2_HC_PROTOCOL_CONTROL_TRANSFER) (  
      IN  EFI_USB2_HC_PROTOCOL  *This,
      IN  UINT8  DeviceAddress,
      IN  UINT8  DeviceSpeed,
      IN  UINTN  MaximumPacketLength,
      IN  EFI_USB_DEVICE_REQUEST  *Request,
      IN  EFI_USB_DATA_DIRECTION  TransferDirection,
      IN OUT VOID  *Data OPTIONAL,
      IN OUT UINTN  *DataLength OPTIONAL,
      IN  UINTN  TimeOut,
      IN  EFI_USB2_HC_TRANSACTION_TRANSLATOR  *Translator,
      OUT  UINT32  *TransferResult
  );

Related Definitions

typedef struct {
    UINT8  TranslatorHubAddress,
    UINT8  TranslatorPortNumber
} EFI_USB2_HC_TRANSACTION_TRANSLATOR;

Parameters

  This   A pointer to the EFI_USB2_HC_PROTOCOL instance. Type EFI_USB2_HC_PROTOCOL is defined in Section 17.1.

  DeviceAddress  Represents the address of the target device on the USB, which is assigned during USB enumeration.

  DeviceSpeed  Indicates device speed. See “Related Definitions” in GetCapability() for a list of the supported values.

  MaximumPacketLength  Indicates the maximum packet size that the default control transfer endpoint is capable of sending or receiving.

  Request  A pointer to the USB device request that will be sent to the USB device. Refer to UsbControlTransfer() (Section 17.2.4) for the definition of this function type.

  TransferDirection  Specifies the data direction for the transfer. There are three values available, EfiUsbDataIn, EfiUsbDataOut and EfiUsbNoData. Refer to UsbControlTransfer() (Section 17.2.4) for the definition of this function type.

  Data  A pointer to the buffer of data that will be transmitted to USB device or received from USB device.

  DataLength  On input, indicates the size, in bytes, of the data buffer specified by Data. On output, indicates the amount of data actually transferred.

  Translator  A pointer to the transaction translator data. See “Description” for the detailed information of this data structure.
TimeOut

Indicates the maximum time, in milliseconds, which the transfer is allowed to complete.

TransferResult

A pointer to the detailed result information generated by this control transfer. Refer to `UsbControlTransfer()` (Section 17.2.4) for transfer result types (`EFI_USB_ERR_x`).

Description

This function is used to submit a control transfer to a target USB device specified by `DeviceAddress`. Control transfers are intended to support configuration/command/status type communication flows between host and USB device.

There are three control transfer types according to the data phase. If the `TransferDirection` parameter is `EfiUsbNoData`, `Data` is `NULL`, and `DataLength` is 0, then no data phase is present in the control transfer. If the `TransferDirection` parameter is `EfiUsbDataOut`, then `Data` specifies the data to be transmitted to the device, and `DataLength` specifies the number of bytes to transfer to the device. In this case, there is an OUT DATA stage followed by a SETUP stage. If the `TransferDirection` parameter is `EfiUsbDataIn`, then `Data` specifies the data to be received from the device, and `DataLength` specifies the number of bytes to receive from the device. In this case there is an IN DATA stage followed by a SETUP stage.

Translator is necessary to perform split transactions on low-speed or full-speed devices connected to a high-speed hub. Such transaction require the device connection information: device address and the port number of the hub that device is connected to. This information is passed through the fields of `EFI_USB2_HC_TRANSACTION_TRANSLATOR` structure. See “Related Definitions” for the structure field names. Translator is passed as `NULL` for the USB1.1 host controllers transfers or when the transfer is requested for high-speed device connected to USB2.0 controller.

If the control transfer has completed successfully, then `EFI_SUCCESS` is returned. If the transfer cannot be completed within the timeout specified by `TimeOut`, then `EFI_TIMEOUT` is returned. If an error other than timeout occurs during the USB transfer, then `EFI_DEVICE_ERROR` is returned and the detailed error code will be returned in the `TransferResult` parameter.

`EFI_INVALID_PARAMETER` is returned if one of the following conditions is satisfied:

- `TransferDirection` is invalid.
- `TransferDirection`, `Data`, and `DataLength` do not match one of the three control transfer types described above.
- `Request` pointer is `NULL`.
- `MaximumPacketLength` is not valid. If `DeviceSpeed` is `EFI_USB_SPEED_LOW`, then `MaximumPacketLength` must be 8. If `DeviceSpeed` is `EFI_USB_SPEED_FULL` or `EFI_USB_SPEED_HIGH`, then `MaximumPacketLength` must be 8, 16, 32, or 64. If `DeviceSpeed` is `EFI_USB_SPEED_SUPER`, then `MaximumPacketLength` must be 512.
- `TransferResult` pointer is `NULL`.
- Translator is `NULL` while the requested transfer requires split transaction. The conditions of the split transactions are described above in “Description” section.
Status Codes Returned

<table>
<thead>
<tr>
<th>Code</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>EFI_SUCCESS</td>
<td>The control transfer was completed successfully.</td>
</tr>
<tr>
<td>EFI_OUT_OF_RESOURCES</td>
<td>The control transfer could not be completed due to a lack of resources.</td>
</tr>
<tr>
<td>EFI_INVALID_PARAMETER</td>
<td>Some parameters are invalid. The possible invalid parameters are described in “Description” above.</td>
</tr>
<tr>
<td>EFI_TIMEOUT</td>
<td>The control transfer failed due to timeout.</td>
</tr>
<tr>
<td>EFI_DEVICE_ERROR</td>
<td>The control transfer failed due to host controller or device error. Caller should check TransferResult for detailed error information.</td>
</tr>
</tbody>
</table>

**EFI_USB2_HC_PROTOCOL.BulkTransfer()**

**Summary**
Submits bulk transfer to a bulk endpoint of a USB device.

**Prototype**
```c
typedef EFI_STATUS (EFIAPI *EFI_USB2_HC_PROTOCOL_BULK_TRANSFER) (  
  IN EFI_USB2_HC_PROTOCOL *This,  
  IN UINT8 DeviceAddress,  
  IN UINT8 EndPointAddress,  
  IN UINT8 DeviceSpeed,  
  IN UINTN MaximumPacketLength,  
  IN UINT8 DataBuffersNumber,  
  IN OUT VOID *Data[EFI_USB_MAX_BULK_BUFFER_NUM],  
  IN OUT UINTN *DataLength,  
  IN OUT UINT8 *DataToggle,  
  IN UINTN TimeOut,  
  IN EFI_USB2_HC_TRANSACTION_TRANSLATOR *Translator,  
  OUT UINT32 *TransferResult  
);  
```

**Parameters**
- **This**
  A pointer to the EFI_USB2_HC_PROTOCOL instance. Type EFI_USB2_HC_PROTOCOL is defined in Section 17.1.
- **DeviceAddress**
  Represents the address of the target device on the USB, which is assigned during USB enumeration.
- **EndPointAddress**
  The combination of an endpoint number and an endpoint direction of the target USB device. Each endpoint address supports data transfer in one direction except the control endpoint (whose default endpoint address is 0). It is the caller’s responsibility to make sure that the EndPointAddress represents a bulk endpoint.
- **DeviceSpeed**
  Indicates device speed. The supported values are EFI_USB_SPEED_FULL, EFI_USB_SPEED_HIGH or EFI_USB_SPEED_SUPER.
MaximumPacketLength
Indicates the maximum packet size the target endpoint is capable of sending or receiving.

DataBuffersNumber
Number of data buffers prepared for the transfer.

Data
Array of pointers to the buffers of data that will be transmitted to USB device or received from USB device.

DataLength
When input, indicates the size, in bytes, of the data buffers specified by Data. When output, indicates the actually transferred data size.

DataToggle
A pointer to the data toggle value. On input, it indicates the initial data toggle value the bulk transfer should adopt; on output, it is updated to indicate the data toggle value of the subsequent bulk transfer.

Translator
A pointer to the transaction translator data. See ControlTransfer() “Description” for the detailed information of this data structure.

TimeOut
Indicates the maximum time, in milliseconds, which the transfer is allowed to complete.

TransferResult
A pointer to the detailed result information of the bulk transfer. Refer to UsbControlTransfer() (Section 17.2.4) for transfer result types (EFI_USB_ERR_x).

Description
This function is used to submit bulk transfer to a target endpoint of a USB device. The target endpoint is specified by DeviceAddress and EndpointAddress. Bulk transfers are designed to support devices that need to communicate relatively large amounts of data at highly variable times where the transfer can use any available bandwidth. Bulk transfers can be used only by full-speed and high-speed devices.

High-speed bulk transfers can be performed using multiple data buffers. The number of buffers that are actually prepared for the transfer is specified by DataBuffersNumber. For full-speed bulk transfers this value is ignored.

Data represents a list of pointers to the data buffers. For full-speed bulk transfers only the data pointed by Data[0] shall be used. For high-speed transfers depending on DataLength there several data buffers can be used. The total number of buffers must not exceed EFI_USB_MAX_BULK_BUFFER_NUM. See “Related Definitions” for the EFI_USB_MAX_BULK_BUFFER_NUM value.

The data transfer direction is determined by the endpoint direction that is encoded in the EndpointAddress parameter. Refer to USB Specification, Revision 2.0 on the Endpoint Address encoding.

The DataToggle parameter is used to track target endpoint’s data sequence toggle bits. The USB provides a mechanism to guarantee data packet synchronization between data transmitter and receiver across multiple transactions. The data packet synchronization is achieved with the data sequence toggle bits and the DATA0/DATA1 PIDs. A bulk endpoint’s toggle sequence is initialized to DATA0 when the endpoint experiences a configuration event. It toggles between DATA0 and DATA1 in each successive data transfer. It is host’s responsibility to track the bulk endpoint’s data toggle sequence and set the correct value for each data packet. The input DataToggle value points to the data toggle value for the first data packet of this bulk transfer; the output DataToggle value points to the data toggle value for
the last successfully transferred data packet of this bulk transfer. The caller should record the data toggle value for use in subsequent bulk transfers to the same endpoint.

If the bulk transfer is successful, then EFI_SUCCESS is returned. If USB transfer cannot be completed within the timeout specified by Timeout, then EFI_TIMEOUT is returned. If an error other than timeout occurs during the USB transfer, then EFI_DEVICE_ERROR is returned and the detailed status code is returned in TransferResult.

EFI_INVALID_PARAMETER is returned if one of the following conditions is satisfied:

- Data is NULL.
- DataLength is 0.
- DeviceSpeed is not valid; the legal values are EFI_USB_SPEED_FULL, EFI_USB_SPEED_HIGH, or EFI_USB_SPEED_SUPER.
- MaximumPacketLength is not valid. The legal value of this parameter is 64 or less for full-speed, 512 or less for high-speed, and 1024 or less for super-speed transactions.
- DataToggle points to a value other than 0 and 1.
- TransferResult is NULL.

Status Codes Returned

<table>
<thead>
<tr>
<th>Code</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>EFI_SUCCESS</td>
<td>The bulk transfer was completed successfully.</td>
</tr>
<tr>
<td>EFI_OUT_OF_RESOURCES</td>
<td>The bulk transfer could not be submitted due to lack of resource.</td>
</tr>
<tr>
<td>EFI_INVALID_PARAMETER</td>
<td>Some parameters are invalid. The possible invalid parameters are described in “Description” above.</td>
</tr>
<tr>
<td>EFI_TIMEOUT</td>
<td>The bulk transfer failed due to timeout.</td>
</tr>
<tr>
<td>EFI_DEVICE_ERROR</td>
<td>The bulk transfer failed due to host controller or device error. Caller should check TransferResult for detailed error information.</td>
</tr>
</tbody>
</table>

EFI_USB2_HC_PROTOCOL.AsyncInterruptTransfer()

Summary

Submits an asynchronous interrupt transfer to an interrupt endpoint of a USB device.
Prototype

```c
typedef EFI_STATUS
  (EFIAPI *EFI_USB2_HC_PROTOCOL_ASYNC_INTERRUPT_TRANSFER) (      
    IN EFI_USB2_HC_PROTOCOL *This,
    IN UINT8 DeviceAddress,
    IN UINT8 EndPointAddress,
    IN UINT8 DeviceSpeed,
    IN UINTN MaximumPacketLength,
    IN BOOLEAN IsNewTransfer,
    IN OUT UINT8 *DataToggle,
    IN UINTN PollingInterval OPTIONAL,
    IN UINTN DataLength OPTIONAL,
    IN EFI_USB2_HC_TRANSACTION_TRANSLATOR *Translator OPTIONAL,
    IN EFI_ASYNC_USB_TRANSFER_CALLBACK CallBackFunction OPTIONAL,
    IN VOID *Context OPTIONAL
  );
```

Parameters

**This** A pointer to the EFI_USB2_HC_PROTOCOL instance. Type EFI_USB2_HC_PROTOCOL is defined in Section 17.1.

**DeviceAddress** Represents the address of the target device on the USB, which is assigned during USB enumeration.

**EndPointAddress** The combination of an endpoint number and an endpoint direction of the target USB device. Each endpoint address supports data transfer in one direction except the control endpoint (whose default endpoint address is zero). It is the caller’s responsibility to make sure that the EndPointAddress represents an interrupt endpoint.

**DeviceSpeed** Indicates device speed. See “Related Definitions” in EFI_USB2_HC_PROTOCOL.ControlTransfer() for a list of the supported values.

**MaximumPacketLength** Indicates the maximum packet size the target endpoint is capable of sending or receiving.

**IsNewTransfer** If TRUE, an asynchronous interrupt pipe is built between the host and the target interrupt endpoint. If FALSE, the specified asynchronous interrupt pipe is canceled. If TRUE, and an interrupt transfer exists for the target endpoint, then EFI_INVALID_PARAMETER is returned.

**DataToggle** A pointer to the data toggle value. On input, it is valid when IsNewTransfer is TRUE, and it indicates the initial data toggle value the asynchronous interrupt transfer should adopt. On output, it is valid when IsNewTransfer is FALSE, and it is updated to indicate the data toggle value of the subsequent asynchronous interrupt transfer.

**PollingInterval** Indicates the interval, in milliseconds, that the asynchronous interrupt transfer is polled. This parameter is required when IsNewTransfer is TRUE.
DataLength: Indicates the length of data to be received at the rate specified by PollingInterval from the target asynchronous interrupt endpoint. This parameter is only required when IsNewTransfer is TRUE.

Translator: A pointer to the transaction translator data.

CallBackFunction: The Callback function. This function is called at the rate specified by PollingInterval. This parameter is only required when IsNewTransfer is TRUE. Refer to UsbAsyncInterruptTransfer() (Section 17.2.4) for the definition of this function type.

Context: The context that is passed to the CallBackFunction. This is an optional parameter and may be NULL.

Description
This function is used to submit asynchronous interrupt transfer to a target endpoint of a USB device. The target endpoint is specified by DeviceAddress and EndpointAddress. In the USB Specification, Revision 2.0, interrupt transfer is one of the four USB transfer types. In the EFI_USB2_HC_PROTOCOL, interrupt transfer is divided further into synchronous interrupt transfer and asynchronous interrupt transfer.

An asynchronous interrupt transfer is typically used to query a device’s status at a fixed rate. For example, keyboard, mouse, and hub devices use this type of transfer to query their interrupt endpoints at a fixed rate. The asynchronous interrupt transfer is intended to support the interrupt transfer type of “submit once, execute periodically.” Unless an explicit request is made, the asynchronous transfer will never retire.

If IsNewTransfer is TRUE, then an interrupt transfer is started at a fixed rate. The rate is specified by PollingInterval, the size of the receive buffer is specified by DataLength, and the callback function is specified by CallBackFunction. Context specifies an optional context that is passed to the CallBackFunction each time it is called. The CallBackFunction is intended to provide a means for the host to periodically process interrupt transfer data.

If IsNewTransfer is TRUE, and an interrupt transfer exists for the target end point, then EFI_INVALID_PARAMETER is returned.

If IsNewTransfer is FALSE, then the interrupt transfer is canceled.

EFI_INVALID_PARAMETER is returned if one of the following conditions is satisfied:

- Data transfer direction indicated by EndPointAddress is other than EfiUsbDataIn.
- IsNewTransfer is TRUE and DataLength is 0.
- IsNewTransfer is TRUE and DataToggle points to a value other than 0 and 1.
- IsNewTransfer is TRUE and PollingInterval is not in the range 1..255.
- IsNewTransfer requested where an interrupt transfer exists for the target end point.
Status Codes Returned

<table>
<thead>
<tr>
<th>Status Code</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>EFI_SUCCESS</td>
<td>The asynchronous interrupt transfer request has been successfully submitted or canceled.</td>
</tr>
<tr>
<td>EFI_INVALID_PARAMETER</td>
<td>Some parameters are invalid. The possible invalid parameters are described in “Description” above. When an interrupt transfer exists for the target endpoint and a new transfer is requested, EFI_INVALID_PARAMETER is returned.</td>
</tr>
<tr>
<td>EFI_OUT_OF_RESOURCES</td>
<td>The request could not be completed due to a lack of resources.</td>
</tr>
</tbody>
</table>

**EFI_USB2_HC_PROTOCOL.SyncInterruptTransfer()**

**Summary**
Submits synchronous interrupt transfer to an interrupt endpoint of a USB device.

**Prototype**
```c
typedef EFI_STATUS
(EFIAPI *EFI_USB2_HC_PROTOCOL_SYNC_INTERRUPT_TRANSFER) ( 
    IN   EFI_USB2_HC_PROTOCOL  *This,
    IN   UINT8                 DeviceAddress,
    IN   UINT8                 EndPointAddress,
    IN   UINT8                 DeviceSpeed,
    IN   UINTN                 MaximumPacketLength,
    IN OUT VOID                *Data,
    IN OUT UINTN               *DataLength,
    IN OUT UINT8               *DataToggle,
    IN   UINTN                 TimeOut,
    IN   EFI_USB2_HC_TRANSACTION_TRANSLATOR *Translator
    OUT  UINT32                *TransferResult
);
```

**Parameters**
- **This** A pointer to the EFI_USB2_HC_PROTOCOL instance. Type EFI_USB2_HC_PROTOCOL is defined in Section 17.1.
- **DeviceAddress** Represents the address of the target device on the USB, which is assigned during USB enumeration.
- **EndPointAddress** The combination of an endpoint number and an endpoint direction of the target USB device. Each endpoint address supports data transfer in one direction except the control endpoint (whose default endpoint address is zero). It is the caller’s responsibility to make sure that the EndPointAddress represents an interrupt endpoint.
- **DeviceSpeed** Indicates device speed. See “Related Definitions” in ControlTransfer() for a list of the supported values.
- **MaximumPacketLength** Indicates the maximum packet size the target endpoint is capable of sending or receiving.
**Data**
A pointer to the buffer of data that will be transmitted to USB device or received from USB device.

**DataLength**
On input, the size, in bytes, of the data buffer specified by **Data**. On output, the number of bytes transferred.

**DataToggle**
A pointer to the data toggle value. On input, it indicates the initial data toggle value the synchronous interrupt transfer should adopt; on output, it is updated to indicate the data toggle value of the subsequent synchronous interrupt transfer.

**TimeOut**
Indicates the maximum time, in milliseconds, which the transfer is allowed to complete.

**Translator**
A pointer to the transaction translator data.

**TransferResult**
A pointer to the detailed result information from the synchronous interrupt transfer. Refer to **UsbControlTransfer()** (Section 17.2.4) for transfer result types (**EFI_USB_ERR_x**).

**Description**
This function is used to submit a synchronous interrupt transfer to a target endpoint of a USB device. The target endpoint is specified by **DeviceAddress** and **EndpointAddress**. In the USB Specification, Revision 2.0, interrupt transfer is one of the four USB transfer types. In the **EFI_USB2_HC_PROTOCOL**, interrupt transfer is divided further into synchronous interrupt transfer and asynchronous interrupt transfer.

The synchronous interrupt transfer is designed to retrieve small amounts of data from a USB device through an interrupt endpoint. A synchronous interrupt transfer is only executed once for each request. This is the most significant difference from the asynchronous interrupt transfer.

If the synchronous interrupt transfer is successful, then **EFI_SUCCESS** is returned. If the USB transfer cannot be completed within the timeout specified by **TimeOut**, then **EFI_TIMEOUT** is returned. If an error other than timeout occurs during the USB transfer, then **EFI_DEVICE_ERROR** is returned and the detailed status code is returned in **TransferResult**.

**EFI_INVALID_PARAMETER** is returned if one of the following conditions is satisfied:

- **Data** is **NULL**.
- **DataLength** is 0.
- **MaximumPacketLength** is not valid. The legal value of this parameter should be 3072 or less for high-speed device, 64 or less for a full-speed device; for a slow device, it is limited to 8 or less. For the full-speed device, it should be 8, 16, 32, or 64; for the slow device, it is limited to 8.
- **DataToggle** points to a value other than 0 and 1.
- **TransferResult** is **NULL**.
Status Codes Returned

<table>
<thead>
<tr>
<th>Code</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>EFI_SUCCESS</td>
<td>The synchronous interrupt transfer was completed successfully.</td>
</tr>
<tr>
<td>EFI_OUT_OF_RESOURCES</td>
<td>The synchronous interrupt transfer could not be submitted due to lack of</td>
</tr>
<tr>
<td></td>
<td>resource.</td>
</tr>
<tr>
<td>EFI_INVALID_PARAMETER</td>
<td>Some parameters are invalid. The possible invalid parameters are</td>
</tr>
<tr>
<td></td>
<td>described in “Description” above.</td>
</tr>
<tr>
<td>EFI_TIMEOUT</td>
<td>The synchronous interrupt transfer failed due to timeout.</td>
</tr>
<tr>
<td>EFI_DEVICE_ERROR</td>
<td>The synchronous interrupt transfer failed due to host controller or device</td>
</tr>
<tr>
<td></td>
<td>error. Caller should check TransferResult for detailed error information.</td>
</tr>
</tbody>
</table>

**EFI_USB2_HC_PROTOCOL.IsochronousTransfer()**

**Summary**

Submits isochronous transfer to an isochronous endpoint of a USB device.

**Prototype**

```c
typedef EFI_STATUS (EFIAPI *EFI_USB2_HC_PROTOCOL_ISOCHRONOUS_TRANSFER) (
  IN   EFI_USB2_HC_PROTOCOL   *This,
  IN   UINT8                 DeviceAddress,
  IN   UINT8                 EndPointAddress,
  IN   UINT8                 DeviceSpeed,
  IN   UINTN                 MaximumPacketLength,
  IN   UINT8                 DataBuffersNumber,
  IN OUT VOID                *Data[EFI_USB_MAX_ISO_BUFFER_NUM],
  IN   UINTN                 DataLength,
  IN   EFI_USB2_HC_TRANSACTION_TRANSLATOR *Translator,
  OUT  UINT32                *TransferResult
);
```

**Related Definitions**

```c
#define EFI_USB_MAX_ISO_BUFFER_NUM   7
#define EFI_USB_MAX_ISO_BUFFER_NUM1  2
```

**Parameters**

- **This**
  
  A pointer to the EFI_USB2_HC_PROTOCOL instance. Type EFI_USB2_HC_PROTOCOL is defined in Section 17.1.

- **DeviceAddress**
  
  Represents the address of the target device on the USB, which is assigned during USB enumeration.

- **EndPointAddress**
  
  The combination of an endpoint number and an endpoint direction of the target USB device. Each endpoint address supports data transfer in one direction except the control endpoint (whose default endpoint address is 0). It is the caller’s responsibility to make sure that the EndPointAddress represents an isochronous endpoint.
DeviceSpeed
Indicates device speed. The supported values are
\texttt{EFI\_USB\_SPEED\_FULL}, \texttt{EFI\_USB\_SPEED\_HIGH}, or
\texttt{EFI\_USB\_SPEED\_SUPER}.

MaximumPacketLength
Indicates the maximum packet size the target endpoint is capable
of sending or receiving. For isochronous endpoints, this value is used
to reserve the bus time in the schedule, required for the per-frame
data payloads. The pipe may, on an ongoing basis, actually use less
bandwidth than that reserved.

DataBuffersNumber
Number of data buffers prepared for the transfer.

Data
Array of pointers to the buffers of data that will be transmitted to
USB device or received from USB device.

DataLength
 Specifies the length, in bytes, of the data to be sent to or received
from the USB device.

Translator
A pointer to the transaction translator data. See ControlTransfer()
“Description” for the detailed information of this data structure.

TransferResult
A pointer to the detail result information of the isochronous
transfer. Refer to \texttt{UsbControlTransfer()} (Section 17.2.4) for
transfer result types (\texttt{EFI\_USB\_ERR\_x}).

\textbf{Description}

This function is used to submit isochronous transfer to a target endpoint of a USB device. The target
endpoint is specified by \texttt{DeviceAddress} and \texttt{EndpointAddress}. Isochronous transfers are used when
working with isochronous data. It provides periodic, continuous communication between the host and a
device. Isochronous transfers can be used only by full-speed, high-speed, and super-speed devices.

High-speed isochronous transfers can be performed using multiple data buffers. The number of buffers
that are actually prepared for the transfer is specified by \texttt{DataBuffersNumber}. For full-speed
isochronous transfers this value is ignored.

Data represents a list of pointers to the data buffers. For full-speed isochronous transfers only the data
pointed by \texttt{Data[0]} shall be used. For high-speed isochronous transfers and for the split transactions
depending on \texttt{DataLength} there several data buffers can be used. For the high-speed isochronous
transfers the total number of buffers must not exceed \texttt{EFI\_USB\_MAX\_ISO\_BUFFER\_NUM}. For split
transactions performed on full-speed device by high-speed host controller the total number of buffers is
limited to \texttt{EFI\_USB\_MAX\_ISO\_BUFFER\_NUM1}. See “Related Definitions” for the
\texttt{EFI\_USB\_MAX\_ISO\_BUFFER\_NUM} and \texttt{EFI\_USB\_MAX\_ISO\_BUFFER\_NUM1} values.

If the isochronous transfer is successful, then \texttt{EFI\_SUCCESS} is returned. The isochronous transfer is
designed to be completed within one USB frame time, if it cannot be completed, \texttt{EFI\_TIMEOUT} is
returned. If an error other than timeout occurs during the USB transfer, then \texttt{EFI\_DEVICE\_ERROR} is
returned and the detailed status code will be returned in \texttt{TransferResult}.

\texttt{EFI\_INVALID\_PARAMETER} is returned if one of the following conditions is satisfied:

- Data is \texttt{NULL}.
- DataLength is 0.
- DeviceSpeed is not one of the supported values listed above.
• **MaximumPacketLength** is invalid. **MaximumPacketLength** must be 1023 or less for full-speed devices, and 1024 or less for high-speed and super-speed devices.

• **TransferResult** is **NULL**.

### Status Codes Returned

<table>
<thead>
<tr>
<th>Status Code</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>EFI_SUCCESS</td>
<td>The isochronous transfer was completed successfully.</td>
</tr>
<tr>
<td>EFI_OUT_OF_RESOURCES</td>
<td>The isochronous transfer could not be submitted due to lack of resource.</td>
</tr>
<tr>
<td>EFI_INVALID_PARAMETER</td>
<td>Some parameters are invalid. The possible invalid parameters are described in “Description” above.</td>
</tr>
<tr>
<td>EFI_TIMEOUT</td>
<td>The isochronous transfer cannot be completed within the one USB frame time.</td>
</tr>
<tr>
<td>EFI_DEVICE_ERROR</td>
<td>The isochronous transfer failed due to host controller or device error. Caller should check <strong>TransferResult</strong> for detailed error information.</td>
</tr>
<tr>
<td>EFI_UNSUPPORTED</td>
<td>The implementation doesn’t support an isochronous transfer function.</td>
</tr>
</tbody>
</table>

### EFI_USB2_HC_PROTOCOL.AsyncIsochronousTransfer()

**Summary**

Submits nonblocking isochronous transfer to an isochronous endpoint of a USB device.

**Prototype**

```c
typedef
 EFI_STATUS
 (EFIAPI * EFI_USB2_HC_PROTOCOL_ASYNC_ISOCHRONOUS_TRANSFER) ( 
  IN   EFI_USB2_HC_PROTOCOL    *This,
  IN   UINT8                   DeviceAddress,
  IN   UINT8                   EndPointAddress,
  IN   UINT8                   DeviceSpeed,
  IN   UINTN                   MaximumPacketLength,
  IN   UINT8                   DataBuffersNumber,
  IN   UINTN                   DataLength,
  IN   EFI_USB2_HC_TRANSACTION_TRANSLATOR  *Translator,
  IN   EFI_ASYNC_USB_TRANSFER_CALLBACK   IsochronousCallBack,
  IN   VOID                     *Context  OPTIONAL
 ) ;
```

**Parameters**

- **This**
  A pointer to the **EFI_USB2_HC_PROTOCOL** instance. Type **EFI_USB2_HC_PROTOCOL** is defined in [Section 17.1](#).

- **DeviceAddress**
  Represents the address of the target device on the USB, which is assigned during USB enumeration.

- **EndPointAddress**
  The combination of an endpoint number and an endpoint direction of the target USB device. Each endpoint address supports data
transfer in one direction except the control endpoint (whose default endpoint address is zero). It is the caller’s responsibility to make sure that the `EndPointAddress` represents an isochronous endpoint.

**DeviceSpeed** Indicates device speed. The supported values are `EFI_USB_SPEED_FULL`, `EFI_USB_SPEED_HIGH`, or `EFI_USB_SPEED_SUPER`.

**MaximumPacketLength** Indicates the maximum packet size the target endpoint is capable of sending or receiving. For isochronous endpoints, this value is used to reserve the bus time in the schedule, required for the per-frame data payloads. The pipe may, on an ongoing basis, actually use less bandwidth than that reserved.

**DataBuffersNumber** Number of data buffers prepared for the transfer.

**Data** Array of pointers to the buffers of data that will be transmitted to USB device or received from USB device.

**DataLength** Specifies the length, in bytes, of the data to be sent to or received from the USB device.

**Translator** A pointer to the transaction translator data. See ControlTransfer() “Description” for the detailed information of this data structure.

**IsochronousCallback** The Callback function. This function is called if the requested isochronous transfer is completed. Refer to `UsbAsyncInterruptTransfer()` (Section 17.2.4) for the definition of this function type.

**Context** Data passed to the `IsochronousCallback` function. This is an optional parameter and may be `NULL`.

**Description**

This is an asynchronous type of USB isochronous transfer. If the caller submits a USB isochronous transfer request through this function, this function will return immediately. When the isochronous transfer completes, the `IsochronousCallback` function will be triggered, the caller can know the transfer results. If the transfer is successful, the caller can get the data received or sent in this callback function.

The target endpoint is specified by `DeviceAddress` and `EndPointAddress`. Isochronous transfers are used when working with isochronous data. It provides periodic, continuous communication between the host and a device. Isochronous transfers can be used only by full-speed, high-speed, and super-speed devices.

High-speed isochronous transfers can be performed using multiple data buffers. The number of buffers that are actually prepared for the transfer is specified by `DataBuffersNumber`. For full-speed isochronous transfers this value is ignored.

Data represents a list of pointers to the data buffers. For full-speed isochronous transfers only the data pointed by `Data[0]` shall be used. For high-speed isochronous transfers and for the split transactions depending on `DataLength` there several data buffers can be used. For the high-speed isochronous transfers the total number of buffers must not exceed `EFI_USB_MAX_ISO_BUFFER_NUM`. For split transactions performed on full-speed device by high-speed host controller the total number of buffers is limited to `EFI_USB_MAX_ISO_BUFFER_NUM1`. See “Related Definitions” in `IsochronousTransfer()` section for the `EFI_USB_MAX_ISO_BUFFER_NUM` and `EFI_USB_MAX_ISO_BUFFER_NUM1` values.
**EFI_INVALID_PARAMETER** is returned if one of the following conditions is satisfied:

- Data is NULL.
- DataLength is 0.
- DeviceSpeed is not one of the supported values listed above.
- MaximumPacketLength is invalid. MaximumPacketLength must be 1023 or less for full-speed devices and 1024 or less for high-speed and super-speed devices.

### Status Codes Returned

<table>
<thead>
<tr>
<th>Status Code</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>EFI_SUCCESS</td>
<td>The asynchronous isochronous transfer was completed successfully.</td>
</tr>
<tr>
<td>EFI_OUT_OF_RESOURCES</td>
<td>The asynchronous isochronous transfer could not be submitted due to lack of resource.</td>
</tr>
<tr>
<td>EFI_INVALID_PARAMETER</td>
<td>Some parameters are invalid. The possible invalid parameters are described in “Description” above.</td>
</tr>
<tr>
<td>EFI_UNSUPPORTED</td>
<td>The implementation doesn’t support Isochronous transfer function</td>
</tr>
</tbody>
</table>

**EFI_USB2_HC_PROTOCOL.GetRootHubPortStatus()**

**Summary**

Retrieves the current status of a USB root hub port.

**Prototype**

```
typedef
EFI_STATUS
(EFIAPI *EFI_USB2_HC_PROTOCOL_GET_ROOTHUB_PORT_STATUS) (  
    IN EFI_USB2_HC_PROTOCOL  *This,
    IN UINT8                  PortNumber,
    OUT EFI_USB_PORT_STATUS   *PortStatus
);
```

**Parameters**

- **This**
  - A pointer to the EFI_USB2_HC_PROTOCOL instance. Type EFI_USB2_HC_PROTOCOL is defined in Section 17.1.

- **PortNumber**
  - Specifies the root hub port from which the status is to be retrieved. This value is zero based. For example, if a root hub has two ports, then the first port is numbered 0, and the second port is numbered 1.

- **PortStatus**
  - A pointer to the current port status bits and port status change bits. The type EFI_USB_PORT_STATUS is defined in “Related Definitions” below.
**Related Definitions**

```c
typedef struct {
    UINT16 PortStatus;
    UINT16 PortChangeStatus;
} EFI_USB_PORT_STATUS;
```

---

**PortStatus**

Contains current port status bitmap. The root hub port status bitmap is unified with the USB hub port status bitmap. See **Table 17-1** for a reference, which is borrowed from *Chapter 11, HubSpecification, of USB Specification, Revision 1.1.*

**PortChangeStatus**

Contains current port status change bitmap. The root hub port change status bitmap is unified with the USB hub port status bitmap. See **Table 17-2** for a reference, which is borrowed from *Chapter 11, Hub Specification, of USB Specification, Revision 1.1.*

---

**Table 17-1 USB Hub Port Status Bitmap**

<table>
<thead>
<tr>
<th>Bit</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td><strong>Current Connect Status:</strong> (USB_PORT_STAT_CONNECTION) This field reflects whether or not a device is currently connected to this port.</td>
</tr>
<tr>
<td></td>
<td>0 = No device is present</td>
</tr>
<tr>
<td></td>
<td>1 = A device is present on this port</td>
</tr>
<tr>
<td>Bit</td>
<td>Description</td>
</tr>
<tr>
<td>-----</td>
<td>-------------</td>
</tr>
</tbody>
</table>
| 1   | **Port Enable / Disabled:** (USB_PORT_STAT_ENABLE) Ports can be enabled by software only. Ports can be disabled by either a fault condition (disconnect event or other fault condition) or by software.  
0 = Port is disabled  
1 = Port is enabled |
| 2   | **Suspend:** (USB_PORT_STAT_SUSPEND) This field indicates whether or not the device on this port is suspended.  
0 = Not suspended  
1 = Suspended |
| 3   | **Over-current Indicator:** (USB_PORT_STAT_OVERCURRENT) This field is used to indicate that the current drain on the port exceeds the specified maximum.  
0 = All no over-current condition exists on this port  
1 = An over-current condition exists on this port |
| 4   | **Reset:** (USB_PORT_STAT_RESET) Indicates whether port is in reset state.  
0 = Port is not in reset state  
1 = Port is in reset state |
| 5-7 | Reserved  
These bits return 0 when read. |
| 8   | **Port Power:** (USB_PORT_STAT_POWER) This field reflects a port’s logical, power control state.  
0 = This port is in the Powered-off state  
1 = This port is not in the Powered-off state |
| 9   | **Low Speed Device Attached:** (USB_PORT_STAT_LOW_SPEED) This is relevant only if a device is attached.  
0 = Full-speed device attached to this port  
1 = Low-speed device attached to this port |
| 10  | **High Speed Device Attached:** (USB_PORT_STAT_HIGH_SPEED) This field indicates whether the connected device is high-speed device  
0 = High-speed device is not attached to this port  
1 = High-speed device attached to this port  
NOTE: this bit has precedence over Bit 9; if set, bit 9 must be ignored. |
| 11  | **Super Speed Device Attached:** (USB_PORT_STAT_SUPER_SPEED) This field indicates whether the connected device is a super-speed device.  
0 = Super-speed device is not attached to this port.  
1 = Super-speed device is attached to this port.  
NOTE: This bit has precedence over Bit 9 and Bit 10; if set bits 9,10 must be ignored. |
| 12  | Reserved.  
Bit returns 0 when read. |
| 13  | The host controller owns the specified port.  
0 = Controller does not own the port.  
1 = Controller owns the port |
| 14-15 | Reserved  
These bits return 0 when read. |
Description

This function is used to retrieve the status of the root hub port specified by PortNumber.

EFI_USB_PORT_STATUS describes the port status of a specified USB port. This data structure is designed to be common to both a USB root hub port and a USB hub port.

The number of root hub ports attached to the USB host controller can be determined with the function GetRootHubPortStatus(). If PortNumber is greater than or equal to the number of ports returned by GetRootHubPortNumber(), then EFI_INVALID_PARAMETER is returned. Otherwise, the status of the USB root hub port is returned in PortStatus, and EFI_SUCCESS is returned.

Status Codes Returned

<table>
<thead>
<tr>
<th>Code</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>Connect Status Change: Indicates a change has occurred in the port’s Current Connect Status.</td>
</tr>
<tr>
<td>1</td>
<td>Port Enable/Disable Change:</td>
</tr>
<tr>
<td>2</td>
<td>Suspend Change:</td>
</tr>
<tr>
<td>3</td>
<td>Over-Current Indicator Change:</td>
</tr>
<tr>
<td>4</td>
<td>Reset Change:</td>
</tr>
<tr>
<td>5-15</td>
<td>Reserved. These bits return 0 when read.</td>
</tr>
</tbody>
</table>

EFI_USB2_HC_PROTOCOL.SetRootHubPortFeature()

Summary

Sets a feature for the specified root hub port.
Prototype

typedef
EFI_STATUS
(EFIAPI *EFI_USB2_HC_PROTOCOL_SET_ROOTHUB_PORT_FEATURE) (  
    IN EFI_USB2_HC_PROTOCOL *This,
    IN UINT8 PortNumber,
    IN EFI_USB_PORT_FEATURE PortFeature
    );

Parameters

This 
A pointer to the EFI_USB2_HC_PROTOCOL instance. Type EFI_USB2_HC_PROTOCOL is defined in Section 17.1.

PortNumber 
Specifies the root hub port whose feature is requested to be set. This value is zero based. For example, if a root hub has two ports, then the first port is number 0, and the second port is numbered 1.

PortFeature 
Indicates the feature selector associated with the feature set request. The port feature indicator is defined in “Related Definitions” and Table 17-3 below.

Related Definitions

typedef enum {
    EfiUsbPortEnable            = 1,
    EfiUsbPortSuspend           = 2,
    EfiUsbPortReset             = 4,
    EfiUsbPortPower             = 8,
    EfiUsbPortOwner             = 13,
    EfiUsbPortConnectChange     = 16,
    EfiUsbPortEnableChange      = 17,
    EfiUsbPortSuspendChange     = 18,
    EfiUsbPortOverCurrentChange = 19,
    EfiUsbPortResetChange       = 20
} EFI_USB_PORT_FEATURE;

The feature values specified in the enumeration variable have special meaning. Each value indicates its bit index in the port status and status change bitmaps, if combines these two bitmaps into a 32-bit bitmap. The meaning of each port feature is listed in Table 17-3.

Table 17-3 USB Port Features

<table>
<thead>
<tr>
<th>Port Feature</th>
<th>For SetRootHubPortFeature</th>
<th>For ClearRootHubPortFeature</th>
</tr>
</thead>
<tbody>
<tr>
<td>EfiUsbPortEnable</td>
<td>Enable the given port of the root hub.</td>
<td>Disable the given port of the root hub.</td>
</tr>
<tr>
<td>EfiUsbPortSuspend</td>
<td>Put the given port into suspend state.</td>
<td>Restore the given port from the previous suspend state.</td>
</tr>
</tbody>
</table>
**Description**

This function sets the feature specified by `PortFeature` for the USB root hub port specified by `PortNumber`. Setting a feature enables that feature or starts a process associated with that feature. For the meanings about the defined features, please refer to Table 17-1 and Table 17-2.

The number of root hub ports attached to the USB host controller can be determined with the function `GetRootHubPortStatus()`. If `PortNumber` is greater than or equal to the number of ports returned by `GetRootHubPortNumber()`, then `EFI_INVALID_PARAMETER` is returned. If `PortFeature` is not `EfiUsbPortEnable`, `EfiUsbPortSuspend`, `EfiUsbPortReset` nor `EfiUsbPortPower`, then `EFI_INVALID_PARAMETER` is returned.

**Status Codes Returned**

<table>
<thead>
<tr>
<th>Code</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>EFI_SUCCESS</td>
<td>The feature specified by <code>PortFeature</code> was set for the USB root hub port specified by <code>PortNumber</code>.</td>
</tr>
<tr>
<td>EFI_INVALID_PARAMETER</td>
<td><code>PortNumber</code> is invalid or <code>PortFeature</code> is invalid for this function.</td>
</tr>
</tbody>
</table>

**Table 17-1**

<table>
<thead>
<tr>
<th>Feature Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>EfiUsbPortReset</td>
<td>Reset the given port of the root hub. Clear the RESET signal for the given port of the root hub.</td>
</tr>
<tr>
<td>EfiUsbPortPower</td>
<td>Power the given port. Shutdown the power from the given port.</td>
</tr>
<tr>
<td>EfiUsbPortOwner</td>
<td>N/A. Releases the port ownership of this port to companion host controller.</td>
</tr>
<tr>
<td>EfiUsbPortConnectChange</td>
<td>N/A. Clear USB_PORT_STAT_C_CONNECTION bit of the given port of the root hub.</td>
</tr>
<tr>
<td>EfiUsbPortEnableChange</td>
<td>N/A. Clear USB_PORT_STAT_C_ENABLE bit of the given port of the root hub.</td>
</tr>
<tr>
<td>EfiUsbPortSuspendChange</td>
<td>N/A. Clear USB_PORT_STAT_C_SUSPEND bit of the given port of the root hub.</td>
</tr>
<tr>
<td>EfiUsbPortOverCurrentChange</td>
<td>N/A. Clear USB_PORT_STAT_C_OVERCURRENT bit of the given port of the root hub.</td>
</tr>
<tr>
<td>EfiUsbPortResetChange</td>
<td>N/A. Clear USB_PORT_STAT_C_RESET bit of the given port of the root hub.</td>
</tr>
</tbody>
</table>
**EFI_USB2_HC_PROTOCOL.ClearRootHubPortFeature()**

**Summary**
Clears a feature for the specified root hub port.

**Prototype**
```c
typedef
EFI_STATUS
(EFIAPI *EFI_USB2_HC_PROTOCOL_ClearRootHubPortFeature)(
    IN EFI_USB2_HC_PROTOCOL    *This,
    IN UINT8                   PortNumber,
    IN EFI_USB_PORT_FEATURE    PortFeature
);
```

**Parameters**
- **This**: A pointer to the EFI_USB2_HC_PROTOCOL instance. Type EFI_USB2_HC_PROTOCOL is defined in Section 17.1.
- **PortNumber**: Specifies the root hub port whose feature is requested to be cleared. This value is zero-based. For example, if a root hub has two ports, then the first port is number 0, and the second port is numbered 1.
- **PortFeature**: Indicates the feature selector associated with the feature clear request. The port feature indicator (EFI_USB_PORT_FEATURE) is defined in the “Related Definitions” section of the SetRootHubPortFeature() function description and in Table 17-3.

**Description**
This function clears the feature specified by PortFeature for the USB root hub port specified by PortNumber. Clearing a feature disables that feature or stops a process associated with that feature. For the meanings about the defined features, refer to Table 17-1 and Table 17-2.

The number of root hub ports attached to the USB host controller can be determined with the function GetRootHubPortStatus(). If PortNumber is greater than or equal to the number of ports returned by GetRootHubPortNumber(), then EFI_INVALID_PARAMETER is returned. If PortFeature is not EfiUsbPortEnable, EfiUsbPortSuspend, EfiUsbPortPower, EfiUsbPortConnectChange, EfiUsbPortResetChange, EfiUsbPortEnableChange, EfiUsbPortSuspendChange, or EfiUsbPortOverCurrentChange, then EFI_INVALID_PARAMETER is returned.
17.2 USB Driver Model

17.2.1 Scope

Section 17.2 describes the USB Driver Model. This includes the behavior of USB Bus Drivers, the behavior of a USB Device Drivers, and a detailed description of the EFI USB I/O Protocol. This document provides enough material to implement a USB Bus Driver, and the tools required to design and implement USB Device Drivers. It does not provide any information on specific USB devices.

The material contained in this section is designed to extend this specification and the UEFI Driver Model in a way that supports USB device drivers and USB bus drivers. These extensions are provided in the form of USB specific protocols. This document provides the information required to implement a USB Bus Driver in system firmware. The document also contains the information required by driver writers to design and implement USB Device Drivers that a platform may need to boot a UEFI-compliant OS.

The USB Driver Model described here is intended to be a foundation on which a USB Bus Driver and a wide variety of USB Device Drivers can be created. USB Driver Model Overview

The USB Driver Stack includes the USB Bus Driver, USB Host Controller Driver, and individual USB device drivers.

Figure 17-2 USB Bus Controller Handle

In the USB Bus Driver Design, the USB Bus Controller is managed by two drivers. One is USB Host Controller Driver, which consumes its parent bus EFI_XYZ_I/O_PROTOCOL, and produces EFI_USB2_HC_PROTOCOL and attaches it to the Bus Controller Handle. The other one is USB Bus Driver, which consumes EFI_USB2_HC_PROTOCOL, and performs bus enumeration. Figure 17-2 shows protocols that are attached to the USB Bus Controller Handle. Detailed descriptions are presented in the following sections.
17.2.2 USB Bus Driver

USB Bus Driver performs periodic Enumeration on the USB Bus. In USB bus enumeration, when a new USB controller is found, the bus driver does some standard configuration for that new controller, and creates a device handle for it. The EFI_USB_IO_PROTOCOL and the EFI DEVICE_PATH_PROTOCOL are attached to the device handle so that the USB controller can be accessed. The USB Bus Driver is also responsible for connecting USB device drivers to USB controllers. When a USB device is detached from a USB bus, the USB bus driver will stop that USB controller, and uninstall the EFI_USB_IO_PROTOCOL and the EFI_DEVICE_PATH_PROTOCOL from that handle. A detailed description is given in Section 17.2.2.3.

17.2.2.1 USB Bus Driver Entry Point

Like all other device drivers, the entry point for a USB Bus Driver attaches the EFI_DRIVER_BINDING_PROTOCOL to image handle of the USB Bus Driver.

17.2.2.2 Driver Binding Protocol for USB Bus Drivers

The Driver Binding Protocol contains three services. These are Supported(), Start(), and Stop(). Supported() tests to see if the USB Bus Driver can manage a device handle. A USB Bus Driver can only manage a device handle that contains EFI_USB2_HC_PROTOCOL.

The general idea is that the USB Bus Driver is a generic driver. Since there are several types of USB Host Controllers, an EFI_USB2_HC_PROTOCOL is used to abstract the host controller interface. Actually, a USB Bus Driver only requires an EFI_USB2_HC_PROTOCOL.

The Start() function tells the USB Bus Driver to start managing the USB Bus. In this function, the USB Bus Driver creates a device handle for the root hub, and creates a timer to monitor root hub connection changes.

The Stop() function tells the USB Bus Driver to stop managing a USB Host Bus Controller. The Stop() function simply deconfigures the devices attached to the root hub. The deconfiguration is a recursive process. If the device to be deconfigured is a USB hub, then all USB devices attached to its downstream ports will be deconfigured first, then itself. If all of the child devices handles have been destroyed then the EFI_USB2_HC_PROTOCOL is closed. Finally, the Stop()unction will then place the USB Host Bus Controller in a quiescent state.

17.2.2.3 USB Hot-Plug Event

Hot-Plug is one of the most important features provided by USB. A USB bus driver implements this feature through two methods. There are two types of hubs defined in the USB specification. One is the USB root hub, which is implemented in the USB Host controller. A timer event is created for the root hub. The other one is a USB Hub. An event is created for each hub that is correctly configured. All these events are associated with the same trigger which is USB bus numerator.

When USB bus enumeration is triggered, the USB Bus Driver checks the source of the event. This is required because the root hub differs from standard USB hub in checking the hub status. The status of a root hub is retrieved through the EFI_USB2_HC_PROTOCOL, and that status of a standard USB hub is retrieved through a USB control transfer. A detailed description of the enumeration process is presented in the next section.
17.2.2.4 USB Bus Enumeration

When the periodic timer or the hubs notify event is signaled, the USB Bus Driver will perform bus numeration.

1. Determine if the event is from the root hub or a standard USB hub.
2. Determine the port on which the connection change event occurred.
3. Determine if it is a connection change or a disconnection change.
4. If a connect change is detected, then a new device has been attached. Perform the following:
   a. Reset and enable that port.
   b. Configure the new device.
   c. Parse the device configuration descriptors; get all of its interface descriptors (i.e., all USB controllers), and configure each interface.
   d. Create a new handle for each interface (USB Controller) within the USB device. Attach the `EFI_DEVICE_PATH_PROTOCOL`, and the `EFI_USB_IO_PROTOCOL` to each handle.
   e. Connect the USB Controller to a USB device driver with the Boot Service `EFI_BOOT_SERVICES.ConnectController()` if applicable.
   f. If the USB Controller is a USB hub, create a Hub notify event which is associated with the USB Bus Enumerator, and submit an Asynchronous Interrupt Transfer Request (See Section 17.2.4).

5. If a disconnect change, then a device has been detached from the USB Bus. Perform the following:
   a. If the device is not a USB Hub, then find and deconfigure the USB Controllers within the device. Then, stop each USB controller with `EFI_BOOT_SERVICES.DisconnectController()`, and uninstall the `EFI_DEVICE_PATH_PROTOCOL` and the `EFI_USB_IO_PROTOCOL` from the controller’s handle. If the `EFI_BOOT_SERVICES.DisconnectController()` call fails this process must be retried on a subsequent timer tick.
   b. If the USB controller is USB hub controller, first find and deconfigure all its downstream USB devices (this is a recursive process, since there may be additional USB hub controllers on the downstream ports), then deconfigure USB hub controller itself.

17.2.3 USB Device Driver

A USB Device Driver manages a USB Controller and produces a device abstraction for use by a preboot application.

17.2.3.1 USB Device Driver Entry Point

Like all other device drivers, the entry point for a USB Device Driver attaches `EFI.Driver_BINDING_PROTOCOL` to image handle of the USB Device Driver.

17.2.3.2 Driver Binding Protocol for USB Device Drivers

The Driver Binding Protocol contains three services. These are `Supported()`, `Start()`, and `Stop()`.
The `Supported()` function tests to see if the USB Device Driver can manage a device handle. This function checks to see if a controller can be managed by the USB Device Driver. This is done by opening the `EFI_USB_IO_PROTOCOL` bus abstraction on the USB Controller handle, and using the services to determine if this USB Controller matches the profile that the USB Device Driver is capable of managing.

The `Start()` function tells the USB Device Driver to start managing a USB Controller. It opens the `EFI_USB_IO_PROTOCOL` instance from the handle for the USB Controller. This protocol instance is used to perform USB packet transmission over the USB bus. For example, if the USB controller is USB keyboard, then the USB keyboard driver would produce and install the `EFI_SIMPLE_TEXT_INPUT_PROTOCOL` to the USB controller handle.

The `Stop()` function tells the USB Device Driver to stop managing a USB Controller. It removes the I/O abstraction protocol instance previously installed in `Start()` from the USB controller handle. It then closes the `EFI_USB_IO_PROTOCOL`.

### 17.2.4 USB I/O Protocol

This section provides a detailed description of the `EFI_USB_IO_PROTOCOL`. This protocol is used by code, typically drivers, running in the EFI boot services environment to access USB devices like USB keyboards, mice and mass storage devices. In particular, functions for managing devices on USB buses are defined here.

The interfaces provided in the `EFI_USB_IO_PROTOCOL` are for performing basic operations to access USB devices. Typically, USB devices are accessed through the four different transfers types:

- **Controller Transfer**: Typically used to configure the USB device into an operation mode.
- **Interrupt Transfer**: Typically used to get periodic small amount of data, like USB keyboard and mouse.
- **Bulk Transfer**: Typically used to transfer large amounts of data like reading blocks from USB mass storage devices.
- **Isochronous Transfer**: Typically used to transfer data at a fixed rate like voice data.

This protocol also provides mechanisms to manage and configure USB devices and controllers.

**`EFI_USB_IO_PROTOCOL`**

**Summary**

Provides services to manage and communicate with USB devices.

**GUID**

```c
#define EFI_USB_IO_PROTOCOL_GUID "\{0x2B2F68D6,0x0CD2,0x44cf,\}
   {0x8E,0x8B,0xBB,0xA2,0x0B,0x1B,0x5B,0x75}\}
```

**Protocol Interface Structure**

```c
typedef struct _EFI_USB_IO_PROTOCOL {
    EFI_USB_IO_CONTROL_TRANSFER       UsbControlTransfer;
    EFI_USB_IO_BULK_TRANSFER          UsbBulkTransfer;
} EFI_USB_IO_PROTOCOL;
```
EFI_USB_IO_ASYNC_INTERRUPT_TRANSFER

EFI_USB_IO_SYNC_INTERRUPT_TRANSFER

EFI_USB_IO_ISOCHRONOUS_TRANSFER

EFI_USB_IO_ASYNC_ISOCHRONOUS_TRANSFER

UsbAsyncIsochronousTransfer;

EFI_USB_IO_GET_DEVICE_DESCRIPTOR

EFI_USB_IO_GET_CONFIG_DESCRIPTOR

EFI_USB_IO_GET_INTERFACE_DESCRIPTOR

EFI_USB_IO_GET_ENDPOINT_DESCRIPTOR

EFI_USB_IO_GET_SUPPORTED_LANGUAGES

EFI_USB_IO_PORT_RESET

} EFI_USB_IO_PROTOCOL;

Parameters

UsbControlTransfer  Accesses the USB Device through USB Control Transfer Pipe. See the

UsbControlTransfer() function description.

UsbBulkTransfer  Accesses the USB Device through USB Bulk Transfer Pipe. See the

UsbBulkTransfer() function description.

UsbAsyncInterruptTransfer  Non-block USB interrupt transfer. See the

UsbAsyncInterruptTransfer() function description.

UsbSyncInterruptTransfer  Accesses the USB Device through USB Synchronous

Interrupt Transfer Pipe. See the UsbSyncInterruptTransfer() function description.

UsbIsochronousTransfer  Accesses the USB Device through USB Isochronous Transfer Pipe. See the

UsbIsochronousTransfer() function description.

UsbAsyncIsochronousTransfer  Nonblock USB isochronous transfer. See the

UsbAsyncIsochronousTransfer() function description.

UsbGetDeviceDescriptor  Retrieves the device descriptor of a USB device. See the

UsbGetDeviceDescriptor() function description.

UsbGetConfigDescriptor  Retrieves the activated configuration descriptor of a USB device. See the

UsbGetConfigDescriptor() function description.

UsbGetInterfaceDescriptor  Retrieves the interface descriptor of a USB Controller. See the

UsbGetInterfaceDescriptor() function description.

UsbGetEndpointDescriptor  Retrieves the endpoint descriptor of a USB Controller. See the

UsbGetEndpointDescriptor() function description.
UsbGetStringDescriptor
Retrieves the string descriptor inside a USB Device. See the
UsbGetStringDescriptor() function description.

UsbGetSupportedLanguages
Retrieves the array of languages that the USB device supports. See
the UsbGetSupportedLanguages() function description.

UsbPortReset
Resets and reconfigures the USB controller. See the
UsbPortReset() function description.

Description
The EFI_USB_IO_PROTOCOL provides four basic transfers types described in the USB 1.1 Specification. These include control transfer, interrupt transfer, bulk transfer and isochronous transfer. The EFI_USB_IO_PROTOCOL also provides some basic USB device/controller management and configuration interfaces. A USB device driver uses the services of this protocol to manage USB devices.

EFI_USB_IO_PROTOCOL.UsbControlTransfer()

Summary
This function is used to manage a USB device with a control transfer pipe. A control transfer is typically used to perform device initialization and configuration.

Prototype

typedef
    EFI_STATUS
        (EFIAPI *EFI_USB_IO_CONTROL_TRANSFER) (  
        IN   EFI_USB_IO_PROTOCOL *This,  
        IN   EFI_USB_DEVICE_REQUEST *Request,  
        IN   EFI_USB_DATA_DIRECTION Direction,  
        IN   UINT32 Timeout,  
        IN OUT VOID *Data OPTIONAL,  
        IN UINTN DataLength OPTIONAL,  
        OUT UINT32 *Status  
        );

Parameters

This
A pointer to the EFI_USB_IO_PROTOCOL instance. Type
EFI_USB_IO_PROTOCOL is defined in Section 17.2.4.

Request
A pointer to the USB device request that will be sent to the USB
device. See “Related Definitions” below.

Direction
Indicates the data direction. See “Related Definitions” below for this
type.

Data
A pointer to the buffer of data that will be transmitted to USB device
or received from USB device.

Timeout
Indicating the transfer should be completed within this time frame.
The units are in milliseconds. If Timeout is 0, then the caller must
wait for the function to be completed until **EFI_SUCCESS** or **EFI_DEVICE_ERROR** is returned.

**DataLength**

The size, in bytes, of the data buffer specified by **Data**.

**Status**

A pointer to the result of the USB transfer.

### Related Definitions

```c
typedef enum {
    EfiUsbDataIn,
    EfiUsbDataOut,
    EfiUsbNoData
} EFI_USB_DATA_DIRECTION;
```

```c
// Error code for USB Transfer Results

#define EFI_USB_NOERROR        0x0000
#define EFI_USB_ERR_NOTEXECUTE 0x0001
#define EFI_USB_ERRSTALL       0x0002
#define EFI_USB_ERRBUFFER      0x0004
#define EFI_USB_ERRBABBLE      0x0008
#define EFI_USB_ERRNAK         0x0010
#define EFI_USB_ERRCRC         0x0020
#define EFI_USB_ERRTIMEOUT     0x0040
#define EFI_USB_ERRBITSTUFF    0x0080
#define EFI_USB_ERRSYSTEM      0x0100
```

```c
typedef struct {
    UINT8 RequestType;
    UINT8 Request;
    UINT16 Value;
    UINT16 Index;
    UINT16 Length;
} EFI_USB_DEVICE_REQUEST;
```

**RequestType**

The field identifies the characteristics of the specific request.

**Request**

This field specifies the particular request.

**Value**

This field is used to pass a parameter to USB device that is specific to the request.

**Index**

This field is also used to pass a parameter to USB device that is specific to the request.

**Length**

This field specifies the length of the data transferred during the second phase of the control transfer. If it is 0, then there is no data phase in this transfer.
Description
This function allows a USB device driver to communicate with the USB device through a Control Transfer. There are three control transfer types according to the data phase. If the Direction parameter is EfiUsbNoData, Data is NULL, and DataLength is 0, then no data phase exists for the control transfer. If the Direction parameter is EfiUsbDataOut, then Data specifies the data to be transmitted to the device, and DataLength specifies the number of bytes to transfer to the device. In this case there is an OUT DATA stage followed by a SETUP stage. If the Direction parameter is EfiUsbDataIn, then Data specifies the data that is received from the device, and DataLength specifies the number of bytes to receive from the device. In this case there is an IN DATA stage followed by a SETUP stage. After the USB transfer has completed successfully, EFI_SUCCESS is returned. If the transfer cannot be completed due to timeout, then EFI_TIMEOUT is returned. If an error other than timeout occurs during the USB transfer, then EFI_DEVICE_ERROR is returned and the detailed status code is returned in Status.

Status Code Returned

<table>
<thead>
<tr>
<th>Status Code</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>EFI_SUCCESS</td>
<td>The control transfer has been successfully executed.</td>
</tr>
<tr>
<td>EFI_INVALID_PARAMETER</td>
<td>The parameter Direction is not valid.</td>
</tr>
<tr>
<td>EFI_INVALID_PARAMETER</td>
<td>Request is NULL.</td>
</tr>
<tr>
<td>EFI_INVALID_PARAMETER</td>
<td>Status is NULL.</td>
</tr>
<tr>
<td>EFI_OUT_OF_RESOURCES</td>
<td>The request could not be completed due to a lack of resources.</td>
</tr>
<tr>
<td>EFI_TIMEOUT</td>
<td>The control transfer fails due to timeout.</td>
</tr>
<tr>
<td>EFI_DEVICE_ERROR</td>
<td>The transfer failed. The transfer status is returned in Status.</td>
</tr>
</tbody>
</table>

**EFI_USB_IO_PROTOCOL.UsbBulkTransfer()**

Summary
This function is used to manage a USB device with the bulk transfer pipe. Bulk Transfers are typically used to transfer large amounts of data to/from USB devices.

Prototype
typedef EFI_STATUS (EFIAPI *EFI_USB_IO_BULK_TRANSFER) (  
  IN EFI_USB_IO_PROTOCOL *This,  
  IN UINT8 DeviceEndpoint,  
  IN OUT VOID *Data,  
  IN OUT UINTN *DataLength,  
  IN UINTN Timeout,  
  OUT UINT32 *Status  
  );

Parameters
**This** A pointer to the EFI_USB_IO_PROTOCOL instance. Type EFI_USB_IO_PROTOCOL is defined in Section 17.2.4.
**DeviceEndpoint**  
The destination USB device endpoint to which the device request is being sent. *DeviceEndpoint* must be between 0x01 and 0x0F or between 0x81 and 0x8F, otherwise *EFI_INVALID_PARAMETER* is returned. If the endpoint is not a BULK endpoint, *EFI_INVALID_PARAMETER* is returned. The MSB of this parameter indicates the endpoint direction. The number “1” stands for an IN endpoint, and “0” stands for an OUT endpoint.

**Data**  
A pointer to the buffer of data that will be transmitted to USB device or received from USB device.

**DataLength**  
On input, the size, in bytes, of the data buffer specified by *Data*. On output, the number of bytes that were actually transferred.

**Timeout**  
Indicating the transfer should be completed within this time frame. The units are in milliseconds. If *Timeout* is 0, then the caller must wait for the function to be completed until *EFI_SUCCESS* or *EFI_DEVICE_ERROR* is returned.

**Status**  
This parameter indicates the USB transfer status.

**Description**  
This function allows a USB device driver to communicate with the USB device through Bulk Transfer. The transfer direction is determined by the endpoint direction. If the USB transfer is successful, then *EFI_SUCCESS* is returned. If USB transfer cannot be completed within the *Timeout* frame, *EFI_TIMEOUT* is returned. If an error other than timeout occurs during the USB transfer, then *EFI_DEVICE_ERROR* is returned and the detailed status code will be returned in the *Status* parameter.

**Status Code Returned**

<table>
<thead>
<tr>
<th>Code</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>EFI_SUCCESS</td>
<td>The bulk transfer has been successfully executed.</td>
</tr>
<tr>
<td>EFI_INVALID_PARAMETER</td>
<td>If <em>DeviceEndpoint</em> is not valid.</td>
</tr>
<tr>
<td>EFI_INVALID_PARAMETER</td>
<td><em>Data</em> is <strong>NULL</strong>.</td>
</tr>
<tr>
<td>EFI_INVALID_PARAMETER</td>
<td><em>DataLength</em> is <strong>NULL</strong>.</td>
</tr>
<tr>
<td>EFI_INVALID_PARAMETER</td>
<td><em>Status</em> is <strong>NULL</strong>.</td>
</tr>
<tr>
<td>EFI_OUT_OF_RESOURCES</td>
<td>The request could not be completed due to a lack of resources.</td>
</tr>
<tr>
<td>EFI_TIMEOUT</td>
<td>The bulk transfer cannot be completed within <em>Timeout</em> timeframe.</td>
</tr>
<tr>
<td>EFI_DEVICE_ERROR</td>
<td>The transfer failed other than timeout, and the transfer status is returned in <em>Status</em>.</td>
</tr>
</tbody>
</table>

**EEI_USB_IO_PROTOCOL.UsbAsyncInterruptTransfer()**

**Summary**  
This function is used to manage a USB device with an interrupt transfer pipe. An Asynchronous Interrupt Transfer is typically used to query a device’s status at a fixed rate. For example, keyboard, mouse, and hub devices use this type of transfer to query their interrupt endpoints at a fixed rate.
Prototype

```c
typedef EFI_STATUS
(EFIAPI *EFI_USB_IO_ASYNC_INTERRUPT_TRANSFER) (  
  IN EFI_USB_IO_PROTOCOL *This,  
  IN UINT8 DeviceEndpoint,  
  IN BOOLEAN IsNewTransfer,  
  IN UINTN PollingInterval OPTIONAL,  
  IN UINTN DataLength OPTIONAL,  
  IN EFI_ASYNC_USB_TRANSFER_CALLBACK InterruptCallBack OPTIONAL,  
  IN VOID *Context OPTIONAL  
);
```

Parameters

- **This** A pointer to the EFI_USB_IO_PROTOCOL instance. Type EFI_USB_IO_PROTOCOL is defined in Section 17.2.4.
- **DeviceEndpoint** The destination USB device endpoint to which the device request is being sent. DeviceEndpoint must be between 0x01 and 0x0F or between 0x81 and 0x8F, otherwise EFI_INVALID_PARAMETER is returned. If the endpoint is not an INTERRUPT endpoint, EFI_INVALID_PARAMETER is returned. The MSB of this parameter indicates the endpoint direction. The number “1” stands for an IN endpoint, and “0” stands for an OUT endpoint.
- **IsNewTransfer** If TRUE, a new transfer will be submitted to USB controller. If FALSE, the interrupt transfer is deleted from the device’s interrupt transfer queue. If TRUE, and an interrupt transfer exists for the target endpoint, then EFI_INVALID_PARAMETER is returned.
- **PollingInterval** Indicates the periodic rate, in milliseconds, that the transfer is to be executed. This parameter is required when IsNewTransfer is TRUE. The value must be between 1 to 255, otherwise EFI_INVALID_PARAMETER is returned. The units are in milliseconds.
- **DataLength** Specifies the length, in bytes, of the data to be received from the USB device. This parameter is only required when IsNewTransfer is TRUE.
- **Context** Data passed to the InterruptCallBack function. This is an optional parameter and may be NULL.
- **InterruptCallBack** The Callback function. This function is called if the asynchronous interrupt transfer is completed. This parameter is required when IsNewTransfer is TRUE. See “Related Definitions” for the definition of this type.
**Related Definitions**

```c
typedef
EFI_STATUS
(EFIAPI * EFI_ASYNC_USB_TRANSFER_CALLBACK) (  
    IN VOID *Data,
    IN UINTN DataLength,
    IN VOID *Context,
    IN UINT32 Status
);
```

- **Data**
  - Data received or sent via the USB Asynchronous Transfer, if the transfer completed successfully.

- **DataLength**
  - The length of Data received or sent via the Asynchronous Transfer, if transfer successfully completes.

- **Context**
  - Data passed from `UsbAsyncInterruptTransfer()` request.

- **Status**
  - Indicates the result of the asynchronous transfer.

**Description**

This function allows a USB device driver to communicate with a USB device with an Interrupt Transfer. Asynchronous Interrupt transfer is different than the other four transfer types because it is a nonblocking transfer. The interrupt endpoint is queried at a fixed rate, and the data transfer direction is always in the direction from the USB device towards the system.

If `IsNewTransfer` is `TRUE`, then an interrupt transfer is started at a fixed rate. The rate is specified by `PollingInterval`, the size of the receive buffer is specified by `DataLength`, and the callback function is specified by `InterruptCallback`. If `IsNewTransfer` is `TRUE`, and an interrupt transfer exists for the target end point, then `EFI_INVALID_PARAMETER` is returned.

If `IsNewTransfer` is `FALSE`, then the interrupt transfer is canceled.

**Status Code Returned**

<table>
<thead>
<tr>
<th>Code</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>EFI_SUCCESS</td>
<td>The asynchronous USB transfer request has been successfully executed.</td>
</tr>
<tr>
<td>EFI_DEVICE_ERROR</td>
<td>The asynchronous USB transfer request failed. When an interrupt transfer exists for the target end point and a new transfer is requested, <code>EFI_INVALID_PARAMETER</code> is returned.</td>
</tr>
</tbody>
</table>

**Examples**

Below is an example of how an asynchronous interrupt transfer is used. The example shows how a USB Keyboard Device Driver can periodically receive data from interrupt endpoint.
EFI_USB_IO_PROTOCOL  *UsbIo;
EFI_STATUS                Status;
USB_KEYBOARD_DEV          *UsbKeyboardDevice;
EFI_USB_INTERRUPT_CALLBACK KeyboardHandle;

...  
Status = UsbIo->UsbAsyncInterruptTransfer(
       UsbIo,
       UsbKeyboardDevice->IntEndpointAddress,
       TRUE,
       UsbKeyboardDevice->IntPollingInterval,
       8,
       KeyboardHandle,
       UsbKeyboardDevice
    );

...

// The following is the InterruptCallback function. If there is
// any results got from Asynchronous Interrupt Transfer,
// this function will be called.

EFI_STATUS KeyboardHandler(
    IN VOID    *Data,
    IN UINTN   DataLength,
    IN VOID    *Context,
    IN UINT32  Result
)
{
    USB_KEYBOARD_DEV  *UsbKeyboardDevice;
    UINTN           I;

    if (EFI_ERROR(Result))
    {
        //
        // Something error during this transfer,
        // just to some recovery work
        //
        ...

        return EFI_DEVICE_ERROR;
    }

    UsbKeyboardDevice = (USB_KEYBOARD_DEV *)Context;
for(I = 0; I < DataLength; I++)
{
    ParsedData(Data[I]);
    ...
}

return EFI_SUCCESS;
}

EFI_USB_IO_PROTOCOL.UsbSyncInterruptTransfer()

Summary
This function is used to manage a USB device with an interrupt transfer pipe. The difference between
UsbAsyncInterruptTransfer() and UsbSyncInterruptTransfer() is that the Synchronous
interrupt transfer will only be executed one time. Once it returns, regardless of its status, the interrupt
request will be deleted in the system.

Prototype
typedef
EFI_STATUS
(EFI_API *EFI_USB_IO_SYNC_INTERRUPT_TRANSFER) (  
    IN   EFI_USB_IO_PROTOCOL  *This,
    IN   UINT8         DeviceEndpoint,
    IN OUT VOID       *Data,
    IN OUT UINTN      *DataLength,
    IN   UINTN         Timeout,
    OUT  UINT32       *Status
);

Parameters
This A pointer to the EFI_USB_IO_PROTOCOL instance. Type
EFI_USB_IO_PROTOCOL is defined in Section 17.2.4.

DeviceEndpoint The destination USB device endpoint to which the device request is
being sent. DeviceEndpoint must be between 0x01 and 0x0F or
between 0x81 and 0x8F, otherwise EFI_INVALID_PARAMETER is
returned. If the endpoint is not an INTERRUPT endpoint,
EFI_INVALID_PARAMETER is returned. The MSB of this parameter
indicates the endpoint direction. The number “1” stands for an IN
endpoint, and “0” stands for an OUT endpoint.

Data A pointer to the buffer of data that will be transmitted to USB device
or received from USB device.

DataLength On input, then size, in bytes, of the buffer Data. On output, the
amount of data actually transferred.

Timeout The time out, in milliseconds, for this transfer. If Timeout is 0, then
the caller must wait for the function to be completed until
EFI_SUCCESS or EFI_DEVICE_ERROR is returned. If the transfer is
not completed in this time frame, then EFI_TIMEOUT is returned.
**Status**

This parameter indicates the USB transfer status.

**Description**

This function allows a USB device driver to communicate with a USB device through a synchronous interrupt transfer. The `UsbSyncInterruptTransfer()` differs from `UsbAsyncInterruptTransfer()` described in the previous section in that it is a blocking transfer request. The caller must wait for the function return, either successfully or unsuccessfully.

**Status Code Returned**

<table>
<thead>
<tr>
<th>Code</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>EFI_SUCCESS</td>
<td>The sync interrupt transfer has been successfully executed.</td>
</tr>
<tr>
<td>EFI_INVALID_PARAMETER</td>
<td>The parameter <code>DeviceEndpoint</code> is not valid.</td>
</tr>
<tr>
<td>EFI_INVALID_PARAMETER</td>
<td>Data is <code>NULL</code>.</td>
</tr>
<tr>
<td>EFI_INVALID_PARAMETER</td>
<td>DataLength is <code>NULL</code>.</td>
</tr>
<tr>
<td>EFI_INVALID_PARAMETER</td>
<td>Status is <code>NULL</code>.</td>
</tr>
<tr>
<td>EFI_OUT_OF_RESOURCES</td>
<td>The request could not be completed due to a lack of resources.</td>
</tr>
<tr>
<td>EFI_TIMEOUT</td>
<td>The transfer cannot be completed within <code>Timeout</code> timeframe.</td>
</tr>
<tr>
<td>EFI_DEVICE_ERROR</td>
<td>The transfer failed other than timeout, and the transfer status is returned in <code>Status</code>.</td>
</tr>
</tbody>
</table>

**EFI_USB_IO_PROTOCOL.UsbIsochronousTransfer()**

**Summary**

This function is used to manage a USB device with an isochronous transfer pipe. An Isochronous transfer is typically used to transfer streaming data.

**Prototype**

```c
typedef EFI_STATUS
    (EFIAPIC * EFI_USB_IO_ISOCHRONOUS_TRANSFER) (
    IN EFI_USB_IO_PROTOCOL *This,
    IN UINT8 DeviceEndpoint,
    IN OUT VOID *Data,
    IN UINTN DataLength,
    OUT UINT32 *Status
    );
```

**Parameters**

- **This**
  
  A pointer to the EFI_USB_IO_PROTOCOL instance. Type `EFI_USB_IO_PROTOCOL` is defined in Section 17.2.4.

- **DeviceEndpoint**
  
  The destination USB device endpoint to which the device request is being sent. `DeviceEndpoint` must be between 0x01 and 0x0F or between 0x81 and 0x8F, otherwise `EFI_INVALID_PARAMETER` is returned. If the endpoint is not an ISOCRONOUS endpoint, `EFI_INVALID_PARAMETER` is returned. The MSB of this parameter
indicates the endpoint direction. The number “1” stands for an IN endpoint, and “0” stands for an OUT endpoint.

**Data**
A pointer to the buffer of data that will be transmitted to USB device or received from USB device.

**DataLength**
The size, in bytes, of the data buffer specified by `Data`.

**Status**
This parameter indicates the USB transfer status.

**Description**
This function allows a USB device driver to communicate with a USB device with an Isochronous Transfer. The type of transfer is different than the other types because the USB Bus Driver will not attempt to perform error recovery if transfer fails. If the USB transfer is completed successfully, then `EFI_SUCCESS` is returned. The isochronous transfer is designed to be completed within 1 USB frame time, if it cannot be completed, `EFI_TIMEOUT` is returned. If the transfer fails due to other reasons, then `EFI_DEVICE_ERROR` is returned and the detailed error status is returned in `Status`. If the data length exceeds the maximum payload per USB frame time, then it is this function’s responsibility to divide the data into a set of smaller packets that fit into a USB frame time. If all the packets are transferred successfully, then `EFI_SUCCESS` is returned.

**Status Code Returned**

<table>
<thead>
<tr>
<th>Status Code Returned</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>EFI_SUCCESS</td>
<td>The isochronous transfer has been successfully executed.</td>
</tr>
<tr>
<td>EFI_INVALID_PARAMETER</td>
<td>The parameter <code>DeviceEndpoint</code> is not valid.</td>
</tr>
<tr>
<td>EFI_OUT_OF_RESOURCES</td>
<td>The request could not be completed due to a lack of resources.</td>
</tr>
<tr>
<td>EFI_TIMEOUT</td>
<td>The transfer cannot be completed within the 1 USB frame time.</td>
</tr>
<tr>
<td>EFI_DEVICE_ERROR</td>
<td>The transfer failed due to the reason other than timeout, The error status is returned in <code>Status</code>.</td>
</tr>
<tr>
<td>EFI_UNSUPPORTED</td>
<td>The implementation doesn’t support an Isochronous transfer function.</td>
</tr>
</tbody>
</table>

**EFI_USB_IO_PROTOCOL.UsbAsyncIsochronousTransfer()**

**Summary**
This function is used to manage a USB device with an isochronous transfer pipe. An asynchronous Isochronous transfer is a nonblocking USB isochronous transfer.
Prototype

typedef
    EFI_STATUS
    (EFIAPI *EFI_USB_IO_ASYNC_ISOCHRONOUS_TRANSFER) ( 
        IN EFI_USB_IO_PROTOCOL       *This,
        IN UINT8                    DeviceEndpoint,
        IN OUT VOID                *Data,
        IN UINTN                   DataLength,
        IN EFI_ASYNC_USB_TRANSFER_CALLBACK IsochronousCallback,
        IN VOID                     *Context OPTIONAL
    );

Parameters

This
    A pointer to the EFI_USB_IO_PROTOCOL instance. Type EFI_USB_IO_PROTOCOL is defined in Section 17.2.4.

DeviceEndpoint
    The destination USB device endpoint to which the device request is being sent. DeviceEndpoint must be between 0x01 and 0x0F or between 0x81 and 0x8F, otherwise EFI_INVALID_PARAMETER is returned. If the endpoint is not an ISOCHRONOUS endpoint, EFI_INVALID_PARAMETER is returned. The MSB of this parameter indicates the endpoint direction. The number “1” stands for an IN endpoint, and “0” stands for an OUT endpoint.

Data
    A pointer to the buffer of data that will be transmitted to USB device or received from USB device.

DataLength
    Specifies the length, in bytes, of the data to be sent to or received from the USB device.

Context
    Data passed to the IsochronousCallback() function. This is an optional parameter and may be NULL.

IsochronousCallback
    The IsochronousCallback() function. This function is called if the requested isochronous transfer is completed. See the “Related Definitions” section of the UsbAsyncInterruptTransfer() function description.

Description

This is an asynchronous type of USB isochronous transfer. If the caller submits a USB isochronous transfer request through this function, this function will return immediately. When the isochronous transfer completes, the IsochronousCallback() function will be triggered, the caller can know the transfer results. If the transfer is successful, the caller can get the data received or sent in this callback function.
Status Code Returned

<table>
<thead>
<tr>
<th>Status Code</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>EFI_SUCCESS</td>
<td>The asynchronous isochronous transfer has been successfully submitted to the system.</td>
</tr>
<tr>
<td>EFI_INVALID_PARAMETER</td>
<td>The parameter DeviceEndpoint is not valid.</td>
</tr>
<tr>
<td>EFI_OUT_OF_RESOURCES</td>
<td>The request could not be submitted due to a lack of resources.</td>
</tr>
<tr>
<td>EFI_UNSUPPORTED</td>
<td>The implementation doesn’t support an asynchronous isochronous transfer function.</td>
</tr>
</tbody>
</table>

EFI_USB_IO_PROTOCOL.UsbGetDeviceDescriptor()

Summary
Retrieves the USB Device Descriptor.

Prototype

```c
typedef
    EFI_STATUS
    (EFIAPI *EFI_USB_IO_GET_DEVICE_DESCRIPTOR) (
    IN  EFI_USB_IO_PROTOCOL   *This,
    OUT EFI_USB_DEVICE_DESCRIPTOR *DeviceDescriptor
    );
```

Parameters

- **This** A pointer to the EFI_USB_IO_PROTOCOL instance. Type EFI_USB_IO_PROTOCOL is defined in Section 17.2.4.
- **DeviceDescriptor** A pointer to the caller allocated USB Device Descriptor. See “Related Definitions” for a detailed description.
Related Definitions

```c
//
// See USB1.1 for detail description.
//
typedef struct {
    UINT8     Length;
    UINT8     DescriptorType;
    UINT16    BcdUSB;
    UINT8     DeviceClass;
    UINT8     DeviceSubClass;
    UINT8     DeviceProtocol;
    UINT8     MaxPacketSize0;
    UINT16    IdVendor;
    UINT16    IdProduct;
    UINT16    BcdDevice;
    UINT8     StrManufacturer;
    UINT8     StrProduct;
    UINT8     StrSerialNumber;
    UINT8     NumConfigurations;
} EFI_USB_DEVICE_DESCRIPTOR;
```

Description

This function is used to retrieve information about USB devices. This information includes the device class, subclass, and the number of configurations the USB device supports. If `DeviceDescriptor` is `NULL`, then `EFI_INVALID_PARAMETER` is returned. If the USB device descriptor is not found, then `EFI_NOT_FOUND` is returned. Otherwise, the device descriptor is returned in `DeviceDescriptor`, and `EFI_SUCCESS` is returned.

Status Code Returned

<table>
<thead>
<tr>
<th>Code</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>EFI_SUCCESS</td>
<td>The device descriptor was retrieved successfully.</td>
</tr>
<tr>
<td>EFI_INVALID_PARAMETER</td>
<td>DeviceDescriptor is NULL.</td>
</tr>
<tr>
<td>EFI_NOT_FOUND</td>
<td>The device descriptor was not found. The device may not be configured.</td>
</tr>
</tbody>
</table>

`EFI_USB_IO_PROTOCOL.UsbGetConfigDescriptor()`

Summary

Retrieves the USB Device Configuration Descriptor.
Prototype

typedef
EFI_STATUS
( EFINIPI *EFI_USB_IO_GET_CONFIG_DESCRIPTOR) ( 
    IN   EFI_USB_IO_PROTOCOL *This,
    OUT EFI_USB_CONFIG_DESCRIPTOR *ConfigurationDescriptor 
);

Parameters

This A pointer to the EFI USB_IO_PROTOCOL instance. Type EFI_USB_IO_PROTOCOL is defined in Section 17.2.4.

ConfigurationDescriptor A pointer to the caller allocated USB Active Configuration Descriptor. See “Related Definitions” for a detailed description.

Related Definitions

//
// See USB1.1 for detail description.
//
typedef struct {
    UINT8 Length;
    UINT8 DescriptorType;
    UINT16 TotalLength;
    UINT8 NumInterfaces;
    UINT8 ConfigurationValue;
    UINT8 Configuration;
    UINT8 Attributes;
    UINT8 MaxPower;
} EFI_USB_CONFIG_DESCRIPTOR;

Description

This function is used to retrieve the active configuration that the USB device is currently using. If ConfigurationDescriptor is NULL, then EFI_INVALID_PARAMETER is returned. If the USB controller does not contain an active configuration, then EFI_NOT_FOUND is returned. Otherwise, the active configuration is returned in ConfigurationDescriptor, and EFI_SUCCESS is returned.
Status Code Returned

<table>
<thead>
<tr>
<th>Status Code</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>EFI_SUCCESS</td>
<td>The active configuration descriptor was retrieved successfully.</td>
</tr>
<tr>
<td>EFI_INVALID_PARAMETER</td>
<td>ConfigurationDescriptor is NULL.</td>
</tr>
<tr>
<td>EFI_NOT_FOUND</td>
<td>An active configuration descriptor cannot be found. The device may not be configured.</td>
</tr>
</tbody>
</table>

**EFI_USB_IO_PROTOCOL.UsbGetInterfaceDescriptor()**

**Summary**
Retrieves the Interface Descriptor for a USB Device Controller. As stated earlier, an interface within a USB device is equivalently to a USB Controller within the current configuration.

**Prototype**
```c
typedef EFI_STATUS (EFIAPI *EFI_USB_IO_GET_INTERFACE_DESCRIPTOR) (
    IN EFI_USB_IO_PROTOCOL *This,
    OUT EFI_USB_INTERFACE_DESCRIPTOR *InterfaceDescriptor
);
```

**Parameters**
- **This**
  A pointer to the EFI_USB_IO_PROTOCOL instance. Type EFI_USB_IO_PROTOCOL is defined in Section 17.2.4.
- **InterfaceDescriptor**
  A pointer to the caller allocated USB Interface Descriptor within the configuration setting. See “Related Definitions” for a detailed description.

**Related Definitions**
```c
//
// See USB1.1 for detail description.
//
typedef struct {
    UINT8 Length;
    UINT8 DescriptorType;
    UINT8 InterfaceNumber;
    UINT8 AlternateSetting;
    UINT8 NumEndpoints;
    UINT8 InterfaceClass;
    UINT8 InterfaceSubClass;
    UINT8 InterfaceProtocol;
    UINT8 Interface;
} EFI_USB_INTERFACE_DESCRIPTOR;
```

**Description**
This function is used to retrieve the interface descriptor for the USB controller. If InterfaceDescriptor is NULL, then EFI_INVALID_PARAMETER is returned. If the USB controller
does not contain an interface descriptor, then `EFI_NOT_FOUND` is returned. Otherwise, the interface
descriptor is returned in `InterfaceDescriptor`, and `EFI_SUCCESS` is returned.

**Status Code Returned**

<table>
<thead>
<tr>
<th>Status Code Returns</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>EFI_SUCCESS</td>
<td>The interface descriptor retrieved successfully.</td>
</tr>
<tr>
<td>EFI_INVALID_PARAMETER</td>
<td>InterfaceDescriptor is <strong>NULL</strong>.</td>
</tr>
<tr>
<td>EFI_NOT_FOUND</td>
<td>The interface descriptor cannot be found. The device may not be correctly configured.</td>
</tr>
</tbody>
</table>

**EFI_USB_IO_PROTOCOL.UsbGetEndpointDescriptor()**

**Summary**
Retrieves an Endpoint Descriptor within a USB Controller.

**Prototype**

```c
typedef EFI_STATUS
(EFI_API *EFI_USB_IO_GET_ENDPOINT_DESCRIPTOR) (
  IN  EFI_USB_IO_PROTOCOL  *This,
  IN  UINT8                EndpointIndex,
  OUT EFI_USB_ENDPOINT_DESCRIPTOR *EndpointDescriptor
);
```

**Parameters**

- **This**
  A pointer to the `EFI_USB_IO_PROTOCOL` instance. Type `EFI_USB_IO_PROTOCOL` is defined in [Section 17.2.4](#).

- **EndpointIndex**
  Indicates which endpoint descriptor to retrieve. The valid range is 0..15.

- **EndpointDescriptor**
  A pointer to the caller allocated USB Endpoint Descriptor of a USB controller. See “Related Definitions” for a detailed description.
Related Definitions

```c
typedef struct {
    UINT8 Length;
    UINT8 DescriptorType;
    UINT8 EndpointAddress;
    UINT8 Attributes;
    UINT16 MaxPacketSize;
    UINT8 Interval;
} EFI_USB_ENDPOINT_DESCRIPTOR;
```

Description

This function is used to retrieve an endpoint descriptor within a USB controller. If `EndpointIndex` is not in the range 0..15, then `EFI_INVALID_PARAMETER` is returned. If `EndpointDescriptor` is `NULL`, then `EFI_INVALID_PARAMETER` is returned. If the endpoint specified by `EndpointIndex` does not exist within the USB controller, then `EFI_NOT_FOUND` is returned. Otherwise, the endpoint descriptor is returned in `EndpointDescriptor`, and `EFI_SUCCESS` is returned.

Status Code Returned

<table>
<thead>
<tr>
<th>Status Code</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>EFI_SUCCESS</td>
<td>The endpoint descriptor was retrieved successfully.</td>
</tr>
<tr>
<td>EFI_INVALID_PARAMETER</td>
<td><code>EndpointIndex</code> is not valid.</td>
</tr>
<tr>
<td>EFI_INVALID_PARAMETER</td>
<td><code>EndpointDescriptor</code> is <code>NULL</code>.</td>
</tr>
<tr>
<td>EFI_NOT_FOUND</td>
<td>The endpoint descriptor cannot be found. The device may not be correctly configured.</td>
</tr>
</tbody>
</table>

Examples

The following code fragment shows how to retrieve all the endpoint descriptors from a USB controller.
EFI_USB_IO_PROTOCOL *UsbIo;
EFI_USB_INTERFACE_DESCRIPTOR InterfaceDesc;
EFI_USB_ENDPOINT_DESCRIPTOR EndpointDesc;
UINTN Index;

Status = UsbIo->GetInterfaceDescriptor (UsbIo, &InterfaceDesc);

for(Index = 0; Index < InterfaceDesc.NumEndpoints; Index++) {
    Status = UsbIo->GetEndpointDescriptor(UsbIo, Index, &EndpointDesc);
}

EFI_USB_IO_PROTOCOL.UsbGetStringDescriptor()

Summary
Retrieves a string stored in a USB Device.

Prototype

typedef
    EFI_STATUS
    (EFI_API *EFI_USB_IO_GET_STRING_DESCRIPTOR) (  
        IN  EFI_USB_IO_PROTOCOL *This,
        IN  UINT16 LangID,
        IN  UINT8 StringID,
        OUT CHAR16 **String
    );

Parameters
This A pointer to the EFI_USB_IO_PROTOCOL instance. Type EFI_USB_IO_PROTOCOL is defined in Section 17.2.4.
LangID The Language ID for the string being retrieved. See the UsbGetSupportedLanguages() function description for a more detailed description.
StringID The ID of the string being retrieved.
String A pointer to a buffer allocated by this function with EFI_BOOT_SERVICES.AllocatePool() to store the string. If this function returns EFI_SUCCESS, it stores the string the caller wants to get. The caller should release the string buffer with EFI_BOOT_SERVICES.FreePool() after the string is not used any more.
Description
This function is used to retrieve strings stored in a USB device. The string to retrieve is identified by a language and an identifier. The language is specified by `LangID`, and the identifier is specified by `StringID`. If the string is found, it is returned in `String`, and `EFI_SUCCESS` is returned. If the string cannot be found, then `EFI_NOT_FOUND` is returned. The string buffer is allocated by this function with `AllocatePool()`. The caller is responsible for calling `FreePool()` for `String` when it is no longer required.

Status Code Returned

<table>
<thead>
<tr>
<th>Status Code</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>EFI_SUCCESS</td>
<td>The string was retrieved successfully.</td>
</tr>
<tr>
<td>EFI_NOT_FOUND</td>
<td>The string specified by <code>LangID</code> and <code>StringID</code> was not found.</td>
</tr>
<tr>
<td>EFI_OUT_OF_RESOURCES</td>
<td>There are not enough resources to allocate the return buffer <code>String</code>.</td>
</tr>
</tbody>
</table>

**EFI_USB_IO_PROTOCOL.UsbGetSupportedLanguages()**

Summary
Retrieves all the language ID codes that the USB device supports.

Prototype

```c
typedef EFI_STATUS (EFIAPI *EFI_USB_IO_GET_SUPPORTED_LANGUAGES)(
    IN EFI_USB_IO_PROTOCOL *This,
    OUT UINT16 *LangIDTable,
    OUT UINT16 *TableSize
);
```

Parameters

- **This**
  A pointer to the `EFI_USB_IO_PROTOCOL` instance. Type `EFI_USB_IO_PROTOCOL` is defined in [Section 17.2.4](#).

- **LangIDTable**
  Language ID for the string the caller wants to get. This is a 16-bit ID defined by Microsoft. This buffer pointer is allocated and maintained by the USB Bus Driver, the caller should not modify its contents.

- **TableSize**
  The size, in bytes, of the table `LangIDTable`.

Description
Retrieves all the language ID codes that the USB device supports.
Status Code Returned

<table>
<thead>
<tr>
<th>Code</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>EFI_SUCCESS</td>
<td>The support languages were retrieved successfully.</td>
</tr>
</tbody>
</table>

**EFI_USB_IO_PROTOCOL.UsbPortReset()**

**Summary**

Resets and reconfigures the USB controller. This function will work for all USB devices except USB Hub Controllers.

**Prototype**

```c
typedef EFI_STATUS (EFIAPI *EFI_USB_IO_PORT_RESET) (
    IN EFI_USB_IO_PROTOCOL *This
);
```

**Parameters**

- `This` A pointer to the EFI_USB_IO_PROTOCOL instance. Type `EFI_USB_IO_PROTOCOL` is defined in Section 17.2.4.

**Description**

This function provides a reset mechanism by sending a RESET signal from the parent hub port. A reconfiguration process will happen (that includes setting the address and setting the configuration). This reset function does not change the bus topology. A USB hub controller cannot be reset using this function, because it would impact the downstream USB devices. So if the controller is a USB hub controller, then `EFI_INVALID_PARAMETER` is returned.

**Status Code Returned**

<table>
<thead>
<tr>
<th>Code</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>EFI_SUCCESS</td>
<td>The USB controller was reset.</td>
</tr>
<tr>
<td>EFI_INVALID_PARAMETER</td>
<td>If the controller specified by <code>This</code> is a USB hub.</td>
</tr>
<tr>
<td>EFI_DEVICE_ERROR</td>
<td>An error occurred during the reconfiguration process.</td>
</tr>
</tbody>
</table>

**17.3 USB Function Protocol**

This section describes the USB Function Protocol, enabling a USB Function device with a UEFI driver that implements the protocol to communicate with a USB Host device.

The USB Function Protocol provides an I/O abstraction for a USB Controller operating in Function mode (also commonly referred to as Device, Peripheral, or Target mode) and the mechanisms by which the USB Function can communicate with the USB Host. It is used by other UEFI drivers or applications to perform data transactions and basic USB controller management over a USB Function port.

This simple protocol only supports USB 2.0 bulk transfers on systems with a single configuration and a single interface. It does not support isochronous or interrupt transfers, alternate interfaces, or USB 3.0 functionality. Future revisions of this protocol may support these or additional features.
EFI_USBFN_IO_PROTOCOL

Summary
Provides basic data transactions and basic USB controller management for a USB Function port.

GUID

```c
// {32D2963A-FE5D-4f30-B633-6E5DC55803CC}
#define EFI_USBFN_IO_PROTOCOL_GUID
{0x32d2963a, 0xfe5d, 0x4f30,
 {0xb6, 0x33, 0x6e, 0x5d, 0xc5, 0x58, 0x3, 0xcc}};
```

Revision Number

```c
#define EFI_USBFN_IO_PROTOCOL_REVISION 0x00010001
```

Protocol Interface Structure

```c
typedef struct _EFI_USBFN_IO_PROTOCOL {
  UINT32 Revision;
  EFI_USBFN_IO_DETECT_PORT DetectPort;
  EFI_USBFN_IO_CONFIGURE_ENABLE_ENDPOINTS ConfigureEnableEndpoints;
  EFI_USBFN_IO_GET_ENDPOINT_MAX_PACKET_SIZE GetEndpointMaxPacketSize;
  EFI_USBFN_IO_GET_DEVICE_INFO GetDeviceInfo;
  EFI_USBFN_IO_GET_VENDOR_ID_PRODUCT_ID GetVendorIdProductId;
  EFI_USBFN_IO_ABORT_TRANSFER AbortTransfer;
  EFI_USBFN_IO_GET_ENDPOINTSTALL_STATE GetEndpointStallState;
  EFI_USBFN_IO_SET_ENDPOINTSTALL_STATE SetEndpointStallState;
  EFI_USBFN_IO_EVENTHANDLER EventHandler;
  EFI_USBFN_IO_TRANSFER Transfer;
  EFI_USBFN_IO_GET_MAXTRANSFERSIZE GetMaxTransferSize;
  EFI_USBFN_IO_ALLOCATE_TRANSFER_BUFFER AllocateTransferBuffer;
  EFI_USBFN_IO_FREE_TRANSFER_BUFFER FreeTransferBuffer;
  EFI_USBFN_IO_START_CONTROLLER StartController;
  EFI_USBFN_IO_STOP_CONTROLLER StopController;
  EFI_USBFN_IO_SET_ENDPOINTPOLICY SetEndpointPolicy;
  EFI_USBFN_IO_GET_ENDPOINTPOLICY GetEndpointPolicy;
} EFI_USBFN_IO_PROTOCOL;
```
Parameters

Revision
The revision to which the EFI_USBFWN_IO_PROTOCOL adheres. All future revisions must be backwards compatible. If a future version is not backwards compatible, a different GUID must be used.

DetectPort
Returns information about the USB port type. See EFI_USBFWN_IO_PROTOCOL.DetectPort(), "Related Definitions" for more details.

ConfigureEnableEndpoints
Initializes all endpoints based on supplied device and configuration descriptors. Enables the device by setting the run/stop bit.

GetEndpointMaxPacketSize
Returns the maximum packet size of the specified endpoint.

GetDeviceInfo
Returns device specific information based on the supplied identifier as a Unicode string.

GetVendorIdProductId
Returns the vendor-id and product-id of the device.

AbortTransfer
Aborts the transfer on the specified endpoint.

GetEndpointStallState
Returns the stall state on the specified endpoint.

SetEndpointStallState
Sets or clears the stall state on the specified endpoint.

EventHandler
This function is called repeatedly to get information on USB bus states, receive-completion and transmit-completion events on the endpoints, and notification on setup packet on endpoint 0.

Transfer
This function handles transferring data to or from the host on the specified endpoint, depending on the direction specified.

GetMaxTransferSize
The maximum supported transfer size in bytes.

AllocateTransferBuffer
Allocates a transfer buffer of the specified size that satisfies the controller requirements.

FreeTransferBuffer
Deallocates the memory allocated for the transfer buffer by EFI_USBFWN_IO_PROTOCOL.AllocateTransferBuffer() function.

StartController
This function initializes the hardware and the internal data structures. The port must not be activated by this function.

StopController
This function disables the device by deactivating the port.

SetEndpointPolicy
This function sets the configuration policy for the specified non-control endpoint. There are a few calling restrictions for this function. See the EFI_USBFWN_IO_PROTOCOL.SetEndpointPolicy() function definition for more details.

GetEndpointPolicy
This function retrieves the configuration policy for the specified non-control endpoint.
Description
This protocol provides basic data transactions and USB controller management for a USB Function port. It provides a lightweight communication mechanism between a USB Host and a USB Function in the UEFI environment.

Like other UEFI device drivers, the entry point for a USB function driver attaches EFI_DRIVER_BINDING_PROTOCOL to image handle of EFI_USBFN_IO_PROTOCOL driver.

The driver binding protocol contains three services, Supported, Start and Stop.

The Supported function must test to see if this driver supports a given controller.

The Start function must supply power to the USB controller if needed, initialize hardware and internal data structures, and then return. The port must not be activated by this function.

The Stop function must disable the USB controller and power it off if needed.

EFI_USBFN_IO_PROTOCOL.DetectPort()

Summary
Returns information about what USB port type was attached.

Prototype
typedef EFI_STATUS (EFIAPI * EFI_USBFN_IO_DETECT_PORT) (IN EFI_USBFN_IO_PROTOCOL *This,
OUT EFI_USBFN_PORT_TYPE *PortType);

Parameters
This A pointer to the EFI_USBFN_IO_PROTOCOL instance.
PortType Returns the USB port type. Refer to the Related Definitions for this function below for details.

Description
Returns information about the USB port type attached. Refer to the "Related Definitions" below for further details.
Status Codes

EFI_SUCCESS The function returned successfully.
EFI_INVALID_PARAMETER A parameter is invalid.
EFI_DEVICE_ERROR The physical device reported an error.
EFI_NOT_READY The physical device is busy or not ready to process this request or there is no USB port attached to the device.

Related Definitions

typedef enum _EFI_USBFN_PORT_TYPE {
  EfiUsbUnknownPort = 0,
  EfiUsbStandardDownstreamPort,
  EfiUsbChargingDownstreamPort,
  EfiUsbDedicatedChargingPort,
  EfiUsbInvalidDedicatedChargingPort
} EFI_USBFN_PORT_TYPE;

Unknown Port Driver internal default port type, this is never returned by the driver with a success status code.

Standard Downstream Port Standard USB host; refer to USB Battery Charging Specification, Revision 1.2 in Section Q.1 for details and the link.

Charging Downstream Port Standard USB host with special charging properties; refer to USB Battery Charging Specification, Revision 1.2, Section Q.1 for the details and link.

Dedicated Charging Port A wall-charger, not USB host; refer to USB Battery Charging Specification, Revision 1.2, Section Q.1 for details and the link.

Invalid Dedicated Charging Port – Neither a USB host nor a dedicated charging port as defined by the USB Battery Charging Specification, Revision 1.2. (See Section Q.1 for details and the link.) An example is a USB charger that raises the voltages on D+/D-, causing the charger to look like an SDP even though it will never issue a setup packet to the upstream facing port.

EFI_USBFN_IO_PROTOCOL.ConfigureEnableEndpoints()

Summary
Configures endpoints based on supplied device and configuration descriptors.

Prototype
typedef
EFI_STATUS
(EIFIAPI * EFI_USBFN_IO_CONFIGURE_ENABLE_ENDPOINTS) (
    IN EFI_USBFIN_IO_PROTOCOL    *This,
    IN EFI_USB_DEVICE_INFO      *DeviceInfo
);  

Parameters
This A pointer to the EFI_USBFIN_IO_PROTOCOL instance.
DeviceInfo A pointer to EFI_USBFIN_DEVICE_INFO instance. Refer to the "Related Definitions" for this function below for details.

Description
Assuming that the hardware has already been initialized, this function configures the endpoints using the device information supplied by DeviceInfo, activates the port, and starts receiving USB events.

This function must ignore the bMaxPacketSize0 field of the Standard Device Descriptor and the wMaxPacketSize field of the Standard Endpoint Descriptor that are made available through DeviceInfo.

Status Codes
EFI_SUCCESS The function returned successfully.
EFI_INVALID_PARAMETER A parameter is invalid.
EFI_DEVICE_ERROR The physical device reported an error.
EFI_NOT_READY The physical device is busy or not ready to process this request.
EFI_OUT_OF_RESOURCES The request could not be completed due to lack of resources.

Related Definitions
typedef struct {
    EFI_USB_INTERFACE_DESCRIPTOR    *InterfaceDescriptor;
    EFI_USB_ENDPOINT_DESCRIPTOR** EndpointDescriptorTable;
} EFI_USB_INTERFACE_INFO;
typedef struct {
    EFI_USB_CONFIG_DESCRIPTOR  *ConfigDescriptor;
    EFI_USB_INTERFACE_INFO    **InterfaceInfoTable;
} EFI_USB_CONFIG_INFO;

typedef struct {
    EFI_USB_DEVICE_DESCRIPTOR *DeviceDescriptor;
    EFI_USB_CONFIG_INFO      **ConfigInfoTable;
} EFI_USB_DEVICE_INFO;

USB_DEVICE_DESCRIPTOR, USB_CONFIG_DESCRIPTOR, USB_INTERFACE_DESCRIPTOR, and USB_ENDPOINT_DESCRIPTOR are defined in Section 17.2.4.

EFI_USBFN_IO_PROTOCOL.GetEndpointMaxPacketSize()

Summary
Returns the maximum packet size of the specified endpoint type for the supplied bus speed.

Prototype

typedef
EFI_STATUS
(EIFIAPI * EFI_USBFN_IO_GET_ENDPOINT_MAXPACKET_SIZE) (  
    IN EFI_USBFN_IO_PROTOCOL   *This,  
    IN EFI_USB_ENDPOINT_TYPE   EndpointType,  
    IN EFI_USB_BUS_SPEED       BusSpeed,  
    OUT UINT16                 *MaxPacketSize
    );

Parameters

This      A pointer to the EFI_USBFN_IO_PROTOCOL instance.
EndDatepointType  Endpoint type as defined as EFI_USB_ENDPOINT_TYPE in the "Related Definitions" for this function below for details.
BusSpeed Bus speed as defined as EFI_USB_BUS_SPEED in the "Related Definitions" for the EventHandle function for details.
MaxPacketSize The maximum packet size, in bytes, of the specified endpoint type.

Description

Returns the maximum packet size of the specified endpoint type for the supplied bus speed. If the BusSpeed is UsbBusSpeedUnknown, the maximum speed the underlying controller supports is assumed.

This protocol currently does not support isochronous or interrupt transfers. Future revisions of this protocol may eventually support it.
Status Codes

<table>
<thead>
<tr>
<th>Code</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>EFI_SUCCESS</td>
<td>The function returned successfully.</td>
</tr>
<tr>
<td>EFI_INVALID_PARAMETER</td>
<td>A parameter is invalid.</td>
</tr>
<tr>
<td>EFI_DEVICE_ERROR</td>
<td>The physical device reported an error.</td>
</tr>
<tr>
<td>EFI_NOT_READY</td>
<td>The physical device is busy or not ready to process this request.</td>
</tr>
</tbody>
</table>

Related Definitions

typedef enum _EFI_USB_ENDPOINT_TYPE
{
    UsbEndpointControl   = 0x00,
    // UsbEndpointIsochronous = 0x01,
    UsbEndpointBulk      = 0x02,
    // UsbEndpointInterrupt = 0x03
} EFI_USB_ENDPOINT_TYPE;

EFI_USBFN_IO_PROTOCOL.GetDeviceInfo()

Summary

Returns device specific information based on the supplied identifier as a Unicode string.

Prototype

typedef EFI_STATUS
(EFIAPIC * EFI_USBFN_IO_GET_DEVICE_INFO) (
    IN EFI_USBFN_IO_PROTOCOL *This,
    IN EFI_USBFN_DEVICE_INFO_ID Id,
    IN OUT UINTN *BufferSize,
    OUT VOID *Buffer OPTIONAL
);

Parameters

This    A pointer to the EFI_USBFN_IO_PROTOCOL instance.
Id      The requested information id. Refer to the "Related Definitions" for this function below for details.
BufferSize On input, the size of the Buffer in bytes. On output, the amount of data returned in Buffer in bytes.
Buffer   A pointer to a buffer to return the requested information as a Unicode string.

Description

Returns device specific information based on the supplied identifier as a Unicode string. If the supplied Buffer isn’t large enough, or is NULL, the method fails with EFI_BUFFER_TOO_SMALL and the required size is returned through BufferSize. All returned strings are in Unicode format.

An Id of EfiUsbDeviceInfoUnknown is treated as an invalid parameter.
Status Codes

- **EFI_SUCCESS**: The function returned successfully.
- **EFI_INVALID_PARAMETER**: One or more of the following conditions is TRUE:
  - BufferSize is NULL.
  - *BufferSize is not 0 and Buffer is NULL.
  - *Id is invalid.
- **EFI_DEVICE_ERROR**: The physical device reported an error.
- **EFI_NOT_READY**: The physical device is busy or not ready to process this request.
- **EFI_BUFFER_TOO_SMALL**: The buffer is too small to hold the buffer.
  - *BufferSize has been updated with the size needed to hold the request string.

Related Definitions

typedef enum _EFI_USBFN_DEVICE_INFO_ID
{
    EfiUsbDeviceInfoUnknown = 0,
    EfiUsbDeviceInfoSerialNumber,
    EfiUsbDeviceInfoManufacturerName,
    EfiUsbDeviceInfoProductName
} EFI_USBFN_DEVICE_INFO_ID;

**EFI_USBFN_IO_PROTOCOL.GetVendorIdProductId()**

**Summary**
Returns the vendor-id and product-id of the device.

**Prototype**

typedef EFI_STATUS
(EFIAPI * EFI_USBFN_IO_GET_VENDOR_ID_PRODUCT_ID) (  
    IN EFI_USBFN_IO_PROTOCOL *This,  
    OUT UINT16 *Vid,  
    OUT UINT16 *Pid  
);

**Parameters**

- **This**: A pointer to the **EFI_USBFN_IO_PROTOCOL** instance.
- **Vid**: Returned vendor-id of the device.
- **Pid**: Returned product-id of the device.

**Description**
Returns vendor-id and product-id of the device.
Status Codes

EFI_SUCCESS The function returned successfully.
EFI_INVALID_PARAMETER A parameter is invalid.
EFI_NOT_FOUND Unable to return the vendor-id or the product-id

Related Definitions
Vendor IDs (VIDs) are 16-bit numbers that represent the device's vendor company and are assigned and maintained by the USB-IF. Product IDs (PIDs) are 16-bit numbers assigned by each vendor to the device.

**EFI_USBFN_IO_PROTOCOL.AbortTransfer()**

**Summary**
Aborts the transfer on the specified endpoint.

**Prototype**

```c
typedef EFI_STATUS
(EFIAPI * EFI_USBFN_IO_ABORT_TRANSFER) (EFI_USBFN_IO_PROTOCOL *This,
IN UINT8 EndpointIndex,
IN EFI_USBFN_ENDPOINT_DIRECTION Direction);
```

**Parameters**

- **This** A pointer to the **EFI_USBFN_IO_PROTOCOL** instance.
- **EndpointIndex** Indicates the endpoint on which the ongoing transfer needs to be canceled.
- **Direction** Direction of the endpoint. Refer to the "Related Definitions" for this function (below) for details.

**Description**
Aborts the transfer on the specified endpoint. This function should fail with **EFI_INVALID_PARAMETER** if the specified direction is incorrect for the endpoint.

**Status Codes**

- EFI_SUCCESS The function returned successfully.
- EFI_INVALID_PARAMETER A parameter is invalid.
- EFI_DEVICE_ERROR The physical device reported an error.
- EFI_NOT_READY The physical device is busy or not ready to process this request.

**Related Definitions**

```c
typedef enum _EFI_USBFN_ENDPOINT_DIRECTION
{
    EfiUsbEndpointDirectionHostOut = 0,
```
EfiUsbEndpointDirectionHostIn,
EfiUsbEndpointDirectionDeviceTx = EfiUsbEndpointDirectionHostIn,
EfiUsbEndpointDirectionDeviceRx = EfiUsbEndpointDirectionHostOut
} EFI_USBFN_ENDPOINT_DIRECTION;

**EFI_USBFN_IO_PROTOCOL.GetEndpointStallState()**

**Summary**
Returns the stall state on the specified endpoint.

**Prototype**

typedef
EFI_STATUS
(EIFIAPI * EFI_USBFN_IO_GET_ENDPOINTSTALL_STATE) (  
IN EFI_USBFN_IO_PROTOCOL *This,
IN UINT8 EndpointIndex,
IN EFI_USBFN_ENDPOINT_DIRECTION Direction,
IN OUT BOOLEAN *State
);

**Parameters**
- **This**
  A pointer to the EFI_USBFN_IO_PROTOCOL instance.
- **EndpointIndex**
  Indicates the endpoint.
- **Direction**
  Direction of the endpoint. Refer to the "Related Definitions" for
  EFI_USBFN_IO_PROTOCOL.AbortTransfer() for details.
- **State**
  Boolean, true value indicates that the endpoint is in a stalled state,
  false otherwise.

**Description**
Returns the stall state on the specified endpoint. This function would fail with
EFI_INVALID_PARAMETER if the specified direction is incorrect for the endpoint.

**Status Codes**
- EFI_SUCCESS
  The function returned successfully.
- EFI_INVALID_PARAMETER
  A parameter is invalid.
- EFI_DEVICE_ERROR
  The physical device reported an error.
- EFI_NOT_READY
  The physical device is busy or not ready to process this request.

**EFI_USBFN_IO_PROTOCOL.SetEndpointStallState()**

**Summary**
Sets or clears the stall state on the specified endpoint.
Prototype

typedef EFI_STATUS
(EIFIAPI * EFI_USBFN_IO_SET_ENDPOINT_STALL_STATE) ( 
IN EFI_USBFN_IO_PROTOCOL        *This,
IN UINT8                   EndpointIndex,
IN EFI_USBFN_ENDPOINT_DIRECTION Direction,
IN BOOLEAN                 State
);

Parameters

This A pointer to the EFI_USBFN_IO_PROTOCOL instance.
EndpointIndex Indicates the endpoint.
Direction Direction of the endpoint. Refer to the "Related Definitions" for the
EFI_USBFN_IO_PROTOCOL.AbortTransfer() function for
details.
State Requested stall state on the specified endpoint. True value causes
the endpoint to stall; false value clears an existing stall.

Description

Sets or clears the stall state on the specified endpoint. This function would fail with
EFI_INVALID_PARAMETER if the specified direction is incorrect for the endpoint.

Status Codes

EFI_SUCCESS The function returned successfully.
EFI_INVALID_PARAMETER A parameter is invalid.
EFI_DEVICE_ERROR The physical device reported an error.
EFI_NOT_READY The physical device is busy or not ready to process this request.
EFI_USBFN_IO_PROTOCOL.EventHandler()

Summary
This function is called repeatedly to get information on USB bus states, receive-completion and transmit-completion events on the endpoints, and notification on setup packet on endpoint 0.

Prototype

```c
typedef
  EFI_STATUS
  (EFIAPI * EFI_USBFN_IO_EVENTHANDLER) ( 
    IN EFI_USBFN_IO_PROTOCOL       *This,
    OUT EFI_USBFN_MESSAGE          *Message,
    IN OUT UINTN                   *PayloadSize,
    OUT EFI_USBFN_MESSAGE_PAYLOAD  *Payload
  );
```

Parameters

- **This** A pointer to the EFI_USBFN_IO_PROTOCOL instance.
- **Message** Indicates the event that initiated this notification. Refer to the "Related Definitions" for this function (below) for all possible types.
- **PayloadSize** On input, the size of the memory pointed by Payload. On output, the amount of data returned in Payload.
- **Payload** A pointer to EFI_USBFN_MESSAGE_PAYLOAD instance to return additional payload for current message. Refer to the "Related Definitions" for this function (below) for details on the type.

Description
This function is called repeatedly to get information on USB bus states, receive-completion and transmit-completion events on the endpoints, and notification on setup packet on endpoint 0. A class driver must call EFI_USBFN_IO_PROTOCOL.EventHandler() repeatedly to receive updates on the transfer status and number of bytes transferred on various endpoints. Refer to Figure 17-3 for details.
A few messages have an associated payload that is returned in the supplied buffer. The following table describes various messages and their payload:

<table>
<thead>
<tr>
<th>Message</th>
<th>Payload</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>EfiUsbMsgSetupPacket</td>
<td>EFI_USB_DEVICE_REQUEST</td>
<td>SETUP packet was received.</td>
</tr>
<tr>
<td>EfiUsbMsgEndpointStatusChangedRx</td>
<td>EFI_USBFN_TRANSFER_RESULT</td>
<td>Some of the requested data has been transmitted to the host. It is the responsibility of the class driver to determine if any remaining data needs to be re-sent. The Buffer supplied to EFI_USBFN_IO_PROTOCOL.Transfer() must be same as the Buffer field of the payload.</td>
</tr>
<tr>
<td>EfiUsbMsgEndpointStatusChangedTx</td>
<td>EFI_USBFN_TRANSFER_RESULT</td>
<td>Some of the requested data has been received from the host. It is the responsibility of the class driver to determine if it needs to wait for any remaining data. The Buffer supplied to EFI_USBFN_IO_PROTOCOL.Transfer() must be same as the Buffer field of the payload.</td>
</tr>
<tr>
<td>EfiUsbMsgBusEventReset</td>
<td>None</td>
<td>A RESET bus event was signaled.</td>
</tr>
<tr>
<td>EfiUsbMsgBusEventDetach</td>
<td>None</td>
<td>A DETACH bus event was signaled.</td>
</tr>
<tr>
<td>EfiUsbMsgBusEventAttach</td>
<td>None</td>
<td>An ATTACH bus event was signaled.</td>
</tr>
<tr>
<td>EfiUsbMsgBusEventSuspend</td>
<td>None</td>
<td>A SUSPEND bus event was signaled.</td>
</tr>
<tr>
<td>EfiUsbMsgBusEventResume</td>
<td>None</td>
<td>A RESUME bus event was signaled.</td>
</tr>
<tr>
<td>EfiUsbMsgBusEventSpeed</td>
<td>EFI_USB_BUS_SPEED</td>
<td>A Bus speed update was signaled.</td>
</tr>
</tbody>
</table>

Table 17-4 Payload-associated Messages and Descriptions

Status Codes

- **EFI_SUCCESS** The function returned successfully.
- **EFI_INVALID_PARAMETER** A parameter is invalid.
- **EFI DEVICE_ERROR** The physical device reported an error.
- **EFI NOT READY** The physical device is busy or not ready to process this request.
- **EFI BUFFER TOO SMALL** The Supplied buffer is not large enough to hold the message payload.
typedef enum _EFI_USBFN_MESSAGE {
    EfiUsbMsgNone = 0,
    EfiUsbMsgSetupPacket,
    EfiUsbMsgEndpointStatusChangedRx,
    EfiUsbMsgEndpointStatusChangedTx,
    EfiUsbMsgBusEventDetach,
    EfiUsbMsgBusEventAttach,
    EfiUsbMsgBusEventReset,
    EfiUsbMsgBusEventSuspend,
    // Indicates that some of the requested data has been received from the host. It is the responsibility of the class driver to determine if it needs to wait for any remaining data. Returned Buffer contains EFI_USBFN_TRANSFER_RESULT struct containing endpoint number, transfer status and count of bytes received.
    EfiUsbMsgEndpointStatusChangedRx,
    // Indicates that some of the requested data has been transmitted to the host. It is the responsibility of the class driver to determine if any remaining data needs to be resent. Returned Buffer contains EFI_USBFN_TRANSFER_RESULT struct containing endpoint number, transfer status and count of bytes sent.
    EfiUsbMsgEndpointStatusChangedTx,
    // DETACH bus event signaled
    EfiUsbMsgBusEventDetach,
    // ATTACH bus event signaled
    EfiUsbMsgBusEventAttach,
    // RESET bus event signaled
    EfiUsbMsgBusEventReset,
    // SUSPEND bus event signaled
}
/
EfiUsbMsgBusEventSuspend,
/
// RESUME bus event signaled
//
EfiUsbMsgBusEventResume,
//
// Bus speed updated, returned buffer indicated bus speed
// using following enumeration named EFI_USB_BUS_SPEED
//
EfiUsbMsgBusEventSpeed
} EFI_USBFN_MESSAGE;

typedef enum _EFI_USBFN_TRANSFER_STATUS {
  UsbTransferStatusUnknown = 0,
  UsbTransferStatusComplete,
  UsbTransferStatusAborted,
  UsbTransferStatusActive,
  UsbTransferStatusNone
} EFI_USBFN_TRANSFER_STATUS;

typedef struct _EFI_USBFN_TRANSFER_RESULT {
  UINTN BytesTransferred;
  EFI_USBFN_TRANSFER_STATUS TransferStatus;
  UINT8 EndpointIndex;
  EFI_USBFN_ENDPOINT_DIRECTION Direction;
 VOID *Buffer;
} EFI_USBFN_TRANSFER_RESULT;

typedef enum _EFI_USB_BUS_SPEED {
  UsbBusSpeedUnknown = 0,
  UsbBusSpeedLow,
  UsbBusSpeedFull,
  UsbBusSpeedHigh,
  UsbBusSpeedSuper,
  UsbBusSpeedMaximum = UsbBusSpeedSuper
} EFI_USB_BUS_SPEED;

typedef union _EFI_USBFN_MESSAGE_PAYLOAD {
  EFI_USB_DEVICE_REQUEST udr;
  EFI_USBFN_TRANSFER_RESULT utr;
} EFI_USBFN_MESSAGE;
EFI_USB_BUS_SPEED ubs;
} EFI_USBFN_MESSAGE_PAYLOAD;

EFI_USBFN_IO_PROTOCOL.Transfer()

Summary
This function handles transferring data to or from the host on the specified endpoint, depending on the direction specified.

Prototype

typedef EFI_STATUS (EFIAPI *EFI_USBFN_IO_TRANSFER) (    
    IN EFI_USBFN_IO_PROTOCOL *This,
    IN UINT8 EndpointIndex,
    IN EFI_USBFN_ENDPOINT_DIRECTION Direction,
    IN OUT UINTN *BufferSize,
    IN OUT VOID *Buffer
    );

Parameters
This A pointer to the EFI_USBFN_IO_PROTOCOL instance.
EndpointIndex Indicates the endpoint on which TX or RX transfer needs to take place.
Direction Direction of the endpoint. Refer to the "Related Definitions" of the EFI_USBFN_IO_PROTOCOL.AbortTransfer() function for details.
BufferSize If Direction is EfiUsbEndpointDirectionDeviceRx: On input, the size of the Buffer in bytes. On output, the amount of data returned in Buffer in bytes.
If Direction is EfiUsbEndpointDirectionDeviceTx: On input, the size of the Buffer in bytes. On output, the amount of data transmitted in bytes.
Buffer If Direction is EfiUsbEndpointDirectionDeviceRx: The Buffer to return the received data.
If Direction is EfiUsbEndpointDirectionDeviceTx: The Buffer that contains the data to be transmitted.

Note: This buffer is allocated and freed using the EFI_USBFN_IO_PROTOCOL.AbortTransfer() and EFI_USBFN_IO_PROTOCOL.FreeTransferBuffer() functions. The caller of this function must not free or reuse the buffer until EfiUsbMsgEndpointStatusChangedRx or EfiUsbMsgEndpointStatusChangedTx message was received along with the address of the transfer buffer as part of the message payload. Refer to the function definition for
**EFI_USBFIN_IO_PROTOCOL.EventHandler()** for more information on various messages and their payloads.

**Description**

This function handles transferring data to or from the host on the specified endpoint, depending on the direction specified.

<table>
<thead>
<tr>
<th>Direction</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>EfiUsbEndpointDirectionDeviceTx</td>
<td>Start a transmit transfer on the specified endpoint and return immediately.</td>
</tr>
<tr>
<td>EfiUsbEndpointDirectionDeviceRx</td>
<td>Start a receive transfer on the specified endpoint and return immediately with available data.</td>
</tr>
</tbody>
</table>

A class driver must call **EFI_USBFIN_IO_PROTOCOL.EventHandler()** repeatedly to receive updates on the transfer status and the number of bytes transferred on various endpoints. Upon an update of the transfer status, the **Buffer** field of the **EFI_USBFIN_TRANSFER_RESULT** structure (as described in the function description for **EFI_USBFIN_IO_PROTOCOL.EventHandler()**) must be initialized with the **Buffer** pointer that was supplied to this method.

The overview of the call sequence is illustrated in the Figure 17-3.

This function should fail with **EFI_INVALID_PARAMETER** if the specified direction is incorrect for the endpoint.

**Status codes**

- **EFI_SUCCESS** The function returned successfully.
- **EFI_INVALID_PARAMETER** A parameter is invalid.
- **EFI_DEVICE_ERROR** The physical device reported an error.
- **EFI_NOT_READY** The physical device is busy or not ready to process this request.

**EFI_USBFIN_IO_PROTOCOL.GetMaxTransferSize()**

**Summary**

Returns the maximum supported transfer size.

**Prototype**

```c
typedef EFI_STATUS
  (EFIAPI * EFI_USBFIN_IO_GET_MAXTRANSFER_SIZE) (IN EFI_USBFIN_IO_PROTOCOL *This,
  OUT UINTN MaxTransferSize);
```

**Parameters**

- **This** A pointer to the **EFI_USBFIN_IO_PROTOCOL** instance.
**MaxTransferSize**  The maximum supported transfer size, in bytes.

**Description**
Returns the maximum number of bytes that the underlying controller can accommodate in a single transfer.

**Status Codes**
- **EFI_SUCCESS**: The function returned successfully.
- **EFI_INVALID_PARAMETER**: A parameter is invalid.
- **EFI_DEVICE_ERROR**: The physical device reported an error.
- **EFI_NOT_READY**: The physical device is busy or not ready to process this request.

**EFI_USBFN_IO_PROTOCOL.AllocateTransferBuffer()**

**Summary**
Allocates a transfer buffer of the specified size that satisfies the controller requirements.

**Prototype**
```
typedef
    EFI_STATUS
    (EFIAPI * EFI_USBFN_IO_ALLOCATE_TRANSFER_BUFFER) (
        IN EFI_USBFN_IO_PROTOCOL  *This,
        IN UINTN                  Size,
        OUT VOID                 **Buffer
    );
```

**Parameters**
- **This**: A pointer to the `EFI_USBFN_IO_PROTOCOL` instance.
- **Size**: The number of bytes to allocate for the transfer buffer.
- **Buffer**: A pointer to a pointer to the allocated buffer if the call succeeds; undefined otherwise.

**Description**
The `AllocateTransferBuffer()` function allocates a memory region of `Size` bytes and returns the address of the allocated memory that satisfies the underlying controller requirements in the location referenced by `Buffer`.

The allocated transfer buffer must be freed using a matching call to `EFI_USBFN_IO_PROTOCOL.FreeTransferBuffer()` function.
Status Codes

EFI_SUCCESS The function returned successfully.
EFI_INVALID_PARAMETER A parameter is invalid.
EFI_OUT_OF_RESOURCES The requested transfer buffer could not be allocated.

EFI_USBFN_IO_PROTOCOL.FreeTransferBuffer()

Summary
Deallocations the memory allocated for the transfer buffer by the
EFI_USBFN_IO_PROTOCOL.AllocateTransferBuffer() function.

Prototype

typedef
EFI_STATUS
(EFIAPI * EFI_USBFN_IO_FREE_TRANSFER_BUFFER) (  
    IN EFI_USBFN_IO_PROTOCOL  *This,
    IN VOID                  *Buffer
);

Parameters

This A pointer to the EFI_USBFN_IO_PROTOCOL instance.
Buffer A pointer to the transfer buffer to deallocate.

Description
The EFI_USBFN_IO_PROTOCOL.FreeTransferBuffer() function deallocates the memory specified by Buffer. The Buffer that is freed must have been allocated by
EFI_USBFN_IO_PROTOCOL.AllocateTransferBuffer().

Status Codes

EFI_SUCCESS The function returned successfully.
EFI_INVALID_PARAMETER A parameter is invalid.

EFI_USBFN_IO_PROTOCOL.StartController()

Summary
This function supplies power to the USB controller if needed and initializes the hardware and the internal data structures. The port must not be activated by this function.
Prototype

typedef

EFI_STATUS

(EIFIAPI * EFI_USBFN_IO_START_CONTROLLER) (IN EFI_USBFN_IO_PROTOCOL *This);

Parameters

This A pointer to the EFI_USBFN_IO_PROTOCOL instance.

Description

This function starts the hardware by supplying power to the USB controller if needed, and initializing the hardware and internal data structures. The port must not be activated by this function.

Status codes

EFI_SUCCESS The function returned successfully.
EFI_INVALID_PARAMETER A parameter is invalid.
EFI_DEVICE_ERROR The physical device reported an error.

EFI_USBFN_IO_PROTOCOL.StopController()

Summary

This function stops the USB hardware device.

Prototype

typedef

EFI_STATUS

(EIFIAPI * EFI_USBFN_IO_STOP_CONTROLLER) (IN EFI_USBFN_IO_PROTOCOL *This);

Parameters

This A pointer to the EFI_USBFN_IO_PROTOCOL instance.

Description

This function stops the USB hardware device
Status codes

- EFI_SUCCESS: The function returned successfully.
- EFI_INVALID_PARAMETER: A parameter is invalid.
- EFI_DEVICE_ERROR: The physical device reported an error.

EFI_USBFN_IO_PROTOCOL.SetEndpointPolicy()

Summary
This function sets the configuration policy for the specified non-control endpoint. Refer to the description for calling restrictions.

Prototype

```c
typedef
EFI_STATUS
(EIFIAPI * EFI_USBFN_SET_ENDPOINT_POLICY) (  
    IN EFI_USBFN_IO_PROTOCOL *This,  
    IN UINT8 EndpointIndex,  
    IN EFI_USBFN_ENDPOINT_DIRECTION Direction,  
    IN EFI_USBFN_POLICY_TYPE PolicyType,  
    IN UINTN BufferSize,  
    IN VOID *Buffer  
);
```

Parameters

- **This**: A pointer to the EFI_USBFN_IO_PROTOCOL instance.
- **EndpointIndex**: Indicates the non-control endpoint for which the policy needs to be set.
- **Direction**: Direction of the endpoint. Refer to the "Related Definitions" for the EFI_USBFN_IO_PROTOCOL.AbortTransfer() function for details.
- **PolicyType**: Policy type the user is trying to set for the specified non-control endpoint. Refer to "Related Definitions" for this function below for details.
- **BufferSize**: The size of the Buffer in bytes.
- **Buffer**: The new value for the policy parameter that PolicyType specifies. Refer to "Related Definitions" for this function below for details.

Description
This function sets the configuration policy for the specified non-control endpoint. This function can only be called before EFI_USBFN_IO_PROTOCOL.StartController() or after EFI_USBFN_IO_PROTOCOL.StopController() has been called.
**Status codes**

<table>
<thead>
<tr>
<th>Status Code</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>EFI_SUCCESS</td>
<td>The function returned successfully.</td>
</tr>
<tr>
<td>EFI_INVALID_PARAMETER</td>
<td>A parameter is invalid.</td>
</tr>
<tr>
<td>EFI_DEVICE_ERROR</td>
<td>The physical device reported an error.</td>
</tr>
<tr>
<td>EFI_UNSUPPORTED</td>
<td>Changing this policy value is not supported.</td>
</tr>
</tbody>
</table>

**Related Definitions**

```c
typedef enum _EFI_USBFN_POLICY_TYPE
{
    EfiUsbPolicyUndefined = 0,
    EfiUsbPolicyMaxTransactionSize,
    EfiUsbPolicyZeroLengthTerminationSupport,
    EfiUsbPolicyZeroLengthTermination
} EFI_USBFN_POLICY_TYPE;
```

**EfiUsbPolicyUndefined**

Invalid policy value that must never be used across driver boundary. If used, the function must not return a success status code.

**EfiUsbPolicyMaxTransactionSize**

EfiUsbPolicyMaxTransactionSize is only used with `EFI_USBFN_IO_PROTOCOL.GetEndpointPolicy()`. It provides the size of the largest single transaction (delivery of service to an endpoint) supported by a controller. It must be greater than or equal to the maximum transfer size that can be retrieved by calling `EFI_USBFN_IO_PROTOCOL.GetMaxTransferSize()`.

<table>
<thead>
<tr>
<th>BufferSize</th>
<th>GetEndpointPolicy BufferSize</th>
<th>SetEndpointPolicy BufferSize</th>
</tr>
</thead>
<tbody>
<tr>
<td>4 bytes, sizeof(UINT32)</td>
<td>EFI_STATUS</td>
<td>EFI_UNSUPPORTED</td>
</tr>
</tbody>
</table>

**EfiUsbPolicyZeroLengthTerminationSupport**

EfiUsbPolicyZeroLengthTerminationSupport is only used with `EFI_USBFN_IO_PROTOCOL.GetEndpointPolicy()`. It is TRUE if the USB controller is capable of automatically handling zero length packets when the transfer size is a multiple of USB maximum packet size and FALSE if it is not supported by the controller.

<table>
<thead>
<tr>
<th>BufferSize</th>
<th>GetEndpointPolicy BufferSize</th>
<th>SetEndpointPolicy BufferSize</th>
</tr>
</thead>
<tbody>
<tr>
<td>1 byte, sizeof (BOOLEAN)</td>
<td>EFI_STATUS</td>
<td>EFI_UNSUPPORTED</td>
</tr>
</tbody>
</table>
EfiUsbPolicyZeroLengthTermination

When used with
EFI_USBFN_IO_PROTOCOL.GetEndpointPolicy(), a TRUE value is returned if the USB controller hardware is configured to automatically handle zero length packets when the transfer size is a multiple of USB maximum packet size; a FALSE value is returned if the controller hardware is not configured to do this.

Using EFI_USBFN_IO_PROTOCOL.SetEndpointPolicy() to set the EfiUsbPolicyZeroLengthTermination policy is only applicable to USB controller hardware capable of supporting automatic zero length packet termination. When this value is set to TRUE, the controller must be configured to handle zero length termination for the specified endpoint. When this value is set to FALSE, the controller must be configured to not handle zero length termination for the specified endpoint.

The USB controller’s default policy must not enable automatic zero length packet termination, even if the hardware is capable of supporting it.

<table>
<thead>
<tr>
<th>GetEndpointPolicy</th>
<th>SetEndpointPolicy</th>
</tr>
</thead>
<tbody>
<tr>
<td>BufferSize</td>
<td>1 byte, sizeof(BOOLEAN)</td>
</tr>
<tr>
<td>Return Status</td>
<td>EFI_STATUS</td>
</tr>
</tbody>
</table>

**EFI_USBFN_IO_PROTOCOL.GetEndpointPolicy()**

**Summary**

This function retrieves the configuration policy for the specified non-control endpoint. There are no associated calling restrictions for this function.

**Prototype**

```c
typedef EFI_STATUS
(EFIAPI * EFI_USBFN_GET_ENDPOINT_POLICY) (  
  IN EFI_USBFN_IO_PROTOCOL        *This,  
  IN UINT8 EndpointIndex,  
  IN EFI_USBFN_ENDPOINT_DIRECTION Direction,  
  IN EFI_USBFN_POLICY_TYPE        PolicyType,  
  IN OUT UINTN                    *BufferSize,  
  IN OUT VOID                     *Buffer  
);  
```

**Parameters**

- **This**: A pointer to the EFI_USBFN_IO_PROTOCOL instance.
- **EndpointIndex**: Indicates the non-control endpoint for which the policy needs to be set.
Direction

Direction of the endpoint. Refer to the "Related Definitions" for the EFI_USBFN_IO_PROTOCOL.AbortTransfer() function for details.

PolicyType

Policy type the user is trying to retrieve for the specified non-control endpoint. Refer to the "Related Definitions" for the EFI_USBFN_IO_PROTOCOL.SetEndpointPolicy() function for details.

BufferSize

On input, the size of Buffer in bytes. On output, the amount of data returned in Buffer in bytes.

Buffer

A pointer to a buffer to return requested endpoint policy value. Refer to the "Related Definitions" for the EFI_USBFN_IO_PROTOCOL.SetEndpointPolicy() function for size requirements of various policy types.

Description

This function retrieves the configuration policy for the specified non-control endpoint. This function has no calling restrictions.

Status codes

EFI_SUCCESS

The function returned successfully.

EFI_INVALID_PARAMETER

A parameter is invalid.

EFI_DEVICE_ERROR

The physical device reported an error.

EFI_UNSUPPORTED

The specified policy value is not supported.

EFI_BUFFER_TOO_SMALL

Supplied buffer is not large enough to hold requested policy value.
USB Function Sequence Diagram

Figure 17-3 Sequence of Operations with Endpoint Policy Changes
18 - Protocols — Debugger Support

This chapter describes a minimal set of protocols and associated data structures necessary to enable the creation of source level debuggers for EFI. It does not fully define a debugger design. Using the services described in this document, it should also be possible to implement a variety of debugger solutions.

18.1 Overview

Efficient UEFI driver and application development requires the availability of source level debugging facilities. Although completely on-target debuggers are clearly possible, UEFI debuggers are generally expected to be remotely hosted. That is to say, the debugger itself will be split between two machines, which are the host and target. A majority of debugger code runs on the host that is typically responsible for disassembly, symbol management, source display, and user interface. Similarly, a smaller piece of code runs on the target that establishes the communication to the host and proxies requests from the host. The on-target code is known as the “debug agent.”

The debug agent design is subdivided further into two parts, which are the processor/platform abstraction and the debugger host specific communication grammar. This specification describes architectural interfaces for the former only. Specific implementations for various debugger host communication grammars can be created that make use of the facilities described in this specification.

The processor/platform abstraction is presented as a pair of protocol interfaces, which are the Debug Support protocol and the Debug Port protocol.

The Debug Support protocol abstracts the processor’s debugging facilities, namely a mechanism to manage the processor’s context via caller-installable exception handlers.

The Debug Port protocol abstracts the device that is used for communication between the host and target. Typically this will be a 16550 serial port, 1394 device, or other device that is nominally a serial stream.

Furthermore, a table driven, quiescent, memory-only mechanism for determining the base address of PE32+ images is provided to enable the debugger host to determine where images are located in memory.

Aside from timing differences that occur because of running code associated with the debug agent and user initiated changes to the machine context, the operation of the on-target debugger component must be transparent to the rest of the system. In addition, no portion of the debug agent that runs in interrupt context may make any calls to EFI services or other protocol interfaces.

The services described in this document do not comprise a complete debugger, rather they provide a minimal abstraction required to implement a wide variety of debugger solutions.

18.2 EFI Debug Support Protocol

This section defines the EFI Debug Support protocol which is used by the debug agent.

18.2.1 EFI Debug Support Protocol Overview

The debug-agent needs to be able to gain control of the machine when certain types of events occur; i.e., breakpoints, processor exceptions, etc. Additionally, the debug agent must also be able to periodically
gain control during operation of the machine to check for asynchronous commands from the host. The EFI Debug Support protocol services enable these capabilities.

The EFI Debug Support protocol interfaces produce callback registration mechanisms which are used by the debug agent to register functions that are invoked either periodically or when specific processor exceptions. When they are invoked by the Debug Support driver, these callback functions are passed the current machine context record. The debug agent may modify this context record to change the machine context which is restored to the machine after the callback function returns. The debug agent does not run in the same context as the rest of UEFI and all modifications to the machine context are deferred until after the callback function returns.

It is expected that there will typically be two instances of the EFI Debug Support protocol in the system. One associated with the native processor instruction set (IA-32, x64, ARM, RISC-V, or Itanium processor family), and one for the EFI virtual machine that implements EFI byte code (EBC).

While multiple instances of the EFI Debug Support protocol are expected, there must never be more than one for any given instruction set.

**EFI_DEBUG_SUPPORT_PROTOCOL**

**Summary**

This protocol provides the services to allow the debug agent to register callback functions that are called either periodically or when specific processor exceptions occur.

**GUID**

```c
#define EFI_DEBUG_SUPPORT_PROTOCOL_GUID
{0x2755590C,0x6F3C,0x42FA,
{0x9E,0xA4,0xA3,0xBA,0x54,0x3C,0xDA,0x25}}
```

**Protocol Interface Structure**

```c
typedef struct {
    EFI_INSTRUCTION_SET_ARCHITECTURE   Isa;
    EFI_GET_MAXIMUM_PROCESSOR_INDEX    GetMaximumProcessorIndex;
    EFI_REGISTER_PERIODIC_CALLBACK    RegisterPeriodicCallback;
    EFI_REGISTER_EXCEPTION_CALLBACK   RegisterExceptionCallback;
    EFI_INVALIDATE_INSTRUCTION_CACHE  InvalidateInstructionCache;
} EFI_DEBUG_SUPPORT_PROTOCOL;
```

**Parameters**

- **Isa**
  
  Declares the processor architecture for this instance of the EFI Debug Support protocol.
GetMaximumProcessorIndex

Returns the maximum processor index value that may be used with EFI_DEBUG_SUPPORT_PROTOCOL.RegisterPeriodicCallback() and EFI_DEBUG_SUPPORT_PROTOCOL.RegisterExceptionCallback(). See the EFI_DEBUG_SUPPORT_PROTOCOL.GetMaximumProcessorIndex() function description.

RegisterPeriodicCallback

Registers a callback function that will be invoked periodically and asynchronously to the execution of EFI. See the RegisterPeriodicCallback() function description.

RegisterExceptionCallback

Registers a callback function that will be called each time the specified processor exception occurs. See the RegisterExceptionCallback() function description.

InvalidateInstructionCache

Invalidate the instruction cache of the processor. This is required by processor architectures where instruction and data caches are not coherent when instructions in the code under debug has been modified by the debug agent. See theEFI_DEBUG_SUPPORT_PROTOCOL.InvalidateInstructionCache() function description.

Related Definitions

Refer to the Microsoft PE/COFF Specification revision 6.2 or later for IMAGE_FILE_MACHINE definitions.

Note: At the time of publication of this specification, the latest revision of the PE/COFF specification was 6.2. The definition of IMAGE_FILE_MACHINE_EBC is not included in revision 6.2 of the PE/COFF specification. It will be added in a future revision of the PE/COFF specification.
// Machine type definition

typedef enum {
    IsaIa32 = IMAGE_FILE_MACHINE_I386,  // 0x014C
    IsaX64 = IMAGE_FILE_MACHINE_X64,     // 0x8664
    IsaIpf = IMAGE_FILE_MACHINE_IA64,    // 0x0200
    IsaEbc = IMAGE_FILE_MACHINE_EBC,     // 0x0EBC
    IsaArm = IMAGE_FILE_MACHINE_ARMTHUMB_MIXED // 0x1C2
    IsaAArch64 = IMAGE_FILE_MACHINE_AARCH64 // 0xAA64
    IsaRISCV32 = IMAGE_FILE_MACHINE_RISCV32 // 0x5032
    IsaRISCV64 = IMAGE_FILE_MACHINE_RISCV64 // 0x5064
    IsaRISCV128 = IMAGE_FILE_MACHINE_RISCV128 // 0x5128
} EFI_INSTRUCTION_SET_ARCHITECTURE;

Description

The EFI Debug Support protocol provides the interfaces required to register debug agent callback functions and to manage the processor’s instruction stream as required. Registered callback functions are invoked in interrupt context when the specified event occurs.

The driver that produces the EFI Debug Support protocol is also responsible for saving the machine context prior to invoking a registered callback function and restoring it after the callback function returns prior to returning to the code under debug. If the debug agent has modified the context record, the modified context must be used in the restore operation.

Furthermore, if the debug agent modifies any of the code under debug (to set a software breakpoint for example), it must call the `InvalidateInstructionCache()` function for the region of memory that has been modified.

**EFI_DEBUG_SUPPORT_PROTOCOL.GetMaximumProcessorIndex()**

**Summary**

Returns the maximum value that may be used for the `ProcessorIndex` parameter in `EFI_DEBUG_SUPPORT_PROTOCOL.RegisterPeriodicCallback()` and `EFI_DEBUG_SUPPORT_PROTOCOL.RegisterExceptionCallback()`.

**Prototype**

```c
typedef
    EFI_STATUS
(EFIAPIC *EFI_GET_MAXIMUM_PROCESSOR_INDEX) ( 
    IN EFI_DEBUG_SUPPORT_PROTOCOL *This,
    OUT UINTN *MaxProcessorIndex
);
```

**Parameters**

- **This**
  A pointer to the `EFI_DEBUG_SUPPORT_PROTOCOL` instance. Type `EFI_DEBUG_SUPPORT_PROTOCOL` is defined in this section.
MaxProcessorIndex  Pointer to a caller-allocated UINTN in which the maximum supported processor index is returned.

Description
The **GetMaximumProcessorIndex()** function returns the maximum processor index in the output parameter MaxProcessorIndex. This value is the largest value that may be used in the ProcessorIndex parameter for both **RegisterPeriodicCallback()** and **RegisterExceptionCallback()**. All values between 0 and MaxProcessorIndex must be supported by **RegisterPeriodicCallback()** and **RegisterExceptionCallback()**.

It is the responsibility of the caller to insure all parameters are correct. There is no provision for parameter checking by **GetMaximumProcessorIndex()**. The implementation behavior when an invalid parameter is passed is not defined by this specification.

Status Codes Returned

<table>
<thead>
<tr>
<th>Status Code</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>EFI_SUCCESS</td>
<td>The function completed successfully.</td>
</tr>
</tbody>
</table>

**EFI_DEBUG_SUPPORT_PROTOCOL.RegisterPeriodicCallback()**

Summary
Registers a function to be called back periodically in interrupt context.

Prototype

```c
typedef EFI_STATUS (EFIAPI *EFI_REGISTER_PERIODIC_CALLBACK) (
    IN EFI_DEBUG_SUPPORT_PROTOCOL  *This,
    IN UINTN                       ProcessorIndex,
    IN EFI_PERIODIC_CALLBACK       PeriodicCallback
); 
```

Parameters

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>This</td>
<td>A pointer to the EFI_DEBUG_SUPPORT_PROTOCOL instance. Type EFI_DEBUG_SUPPORT_PROTOCOL is defined in Section 18.2.</td>
</tr>
<tr>
<td>ProcessorIndex</td>
<td>Specifies which processor the callback function applies to.</td>
</tr>
<tr>
<td>PeriodicCallback</td>
<td>A pointer to a function of type <strong>PERIODIC_CALLBACK</strong> that is the main periodic entry point of the debug agent. It receives as a parameter a pointer to the full context of the interrupted execution thread.</td>
</tr>
</tbody>
</table>
typedef VOID (*EFI_PERIODIC_CALLBACK) (IN OUT EFI_SYSTEM_CONTEXT SystemContext);

// Universal EFI_SYSTEM_CONTEXT definition
typedef union {
    EFI_SYSTEM_CONTEXT_EBC *SystemContextEbc;
    EFI_SYSTEM_CONTEXT_IA32 *SystemContextIa32;
    EFI_SYSTEM_CONTEXT_X64 *SystemContextX64;
    EFI_SYSTEM_CONTEXT_IPF *SystemContextIpf;
    EFI_SYSTEM_CONTEXT_ARM *SystemContextArm;
    EFI_SYSTEM_CONTEXT_AARCH64 *SystemContextAArch64;
    EFI_SYSTEM_CONTEXT_RISCV32 *SystemContextRiscV32;
    EFI_SYSTEM_CONTEXT_RISCV64 *SystemContextRiscV64;
    EFI_SYSTEM_CONTEXT_RISCV128 *SystemContextRiscV128;
} EFI_SYSTEM_CONTEXT;

// System context for virtual EBC processors
typedef struct {
    UINT64 R0, R1, R2, R3, R4, R5, R6, R7;
    UINT64 Flags;
    UINT64 ControlFlags;
    UINT64 Ip;
} EFI_SYSTEM_CONTEXT_EBC;

// System context for RISC-V 32
typedef struct {
    // Integer registers
    UINT32 Zero, Ra, Sp, Gp, Tp, T0, T1, T2;
    UINT32 S0FP, S1, A0, A1, A2, A3, A4, A5, A6, A7;
    UINT32 S2, S3, S4, S5, S6, S7, S8, S9, S10, S11;
    UINT32 T3, T4, T5, T6;

    // Floating registers for F, D and Q Standard Extensions
    UINT128 Ft0, Ft1, Ft2, Ft3, Ft4, Ft5, Ft6, Ft7;
    UINT128 Fs0, Fs1, Fa0, Fa1, Fa2, Fa3, Fa4, Fa5, Fa6, Fa7;
    UINT128 Fs2, Fs3, Fs4, Fs5, Fs6, Fs7, Fs8, Fs9, Fs10, Fs11;
    UINT128 Ft8, Ft9, Ft10, Ft11;
} EFI_SYSTEM_CONTEXT_RISCV32;
// System context for RISC-V 64
typedef struct {
    // Integer registers
    UINT64 Zero, Ra, Sp, Gp, Tp, T0, T1, T2;
    UINT64 S0FP, S1, A0, A1, A2, A3, A4, A5, A6, A7;
    UINT64 S2, S3, S4, S5, S6, S7, S8, S9, S10, S11;
    UINT64 T3, T4, T5, T6;

    // Floating registers for F, D and Q Standard Extensions
    UINT128 Ft0, Ft1, Ft2, Ft3, Ft4, Ft5, Ft6, Ft7;
    UINT128 Fs0, Fs1, Fa0, Fa1, Fa2, Fa3, Fa4, Fa5, Fa6, Fa7;
    UINT128 Fs2, Fs3, Fs4, Fs5, Fs6, Fs7, Fs8, Fs9, Fs10, Fs11;
    UINT128 Ft8, Ft9, Ft10, Ft11;
} EFI_SYSTEM_CONTEXT_RISCV64;

// System context for RISC-V 128
typedef struct {
    // Integer registers
    UINT128 Zero, Ra, Sp, Gp, Tp, T0, T1, T2;
    UINT128 S0FP, S1, A0, A1, A2, A3, A4, A5, A6, A7;
    UINT128 S2, S3, S4, S5, S6, S7, S8, S9, S10, S11;
    UINT128 T3, T4, T5, T6;

    // Floating registers for F, D and Q Standard Extensions
    UINT128 Ft0, Ft1, Ft2, Ft3, Ft4, Ft5, Ft6, Ft7;
    UINT128 Fs0, Fs1, Fa0, Fa1, Fa2, Fa3, Fa4, Fa5, Fa6, Fa7;
    UINT128 Fs2, Fs3, Fs4, Fs5, Fs6, Fs7, Fs8, Fs9, Fs10, Fs11;
    UINT128 Ft8, Ft9, Ft10, Ft11;
} EFI_SYSTEM_CONTEXT_RISCV128;

Note: When the context record field is larger than the register being stored in it, the upper bits of the context record field are unused and ignored
// System context for IA-32 processors
typedef struct {
    UINT32 ExceptionData; // ExceptionData is additional data pushed
    // on the stack by some types of IA-32
    // exceptions
    EFI_FX_SAVE_STATE_IA32 FxSaveState;
    UINT32 Dr0, Dr1, Dr2, Dr3, Dr6, Dr7;
    UINT32 Cr0, Cr1 /* Reserved */, Cr2, Cr3, Cr4;
    UINT32 Eflags;
    UINT32 Ldtr, Tr;
    UINT32 Gdtr[2], Idtr[2];
    UINT32 Eip;
    UINT32 Gs, Fs, Es, Ds, Cs, Ss;
    UINT32 Edi, Esi, Ebp, Esp, Ebx, Edx, Ecx, Eax;
} EFI_SYSTEM_CONTEXT_IA32;

// FXSAVE_STATE - FP / MMX / XMM registers
typedef struct {
    UINT16 Fcw;
    UINT16 Fsw;
    UINT16 Ftw;
    UINT16 Opcode;
    UINT32 Eip;
    UINT16 Cs;
    UINT16 Reserved1;
    UINT32 DataOffset;
    UINT16 Ds;
    UINT8 Reserved2[10];
    UINT8 St0Mm0[10], Reserved3[6];
    UINT8 St1Mm1[10], Reserved4[6];
    UINT8 St2Mm2[10], Reserved5[6];
    UINT8 St3Mm3[10], Reserved6[6];
    UINT8 St4Mm4[10], Reserved7[6];
    UINT8 St5Mm5[10], Reserved8[6];
    UINT8 St6Mm6[10], Reserved9[6];
    UINT8 St7Mm7[10], Reserved10[6];
    UINT8 Xmm0[16];
    UINT8 Xmm1[16];
    UINT8 Xmm2[16];
    UINT8 Xmm3[16];
    UINT8 Xmm4[16];
    UINT8 Xmm5[16];
    UINT8 Xmm6[16];
    UINT8 Xmm7[16];
    UINT8 Reserved11[14 * 16];
} EFI_FX_SAVE_STATE_IA32
// System context for x64 processors
typedef struct {
    UINT64          ExceptionData; // ExceptionData is
    // additional data pushed
    // on the stack by some
    // types of x64 64-bit
    // mode exceptions
    EFI_FX_SAVE_STATE_X64      FxSaveState;
    UINT64        Dr0, Dr1, Dr2, Dr3, Dr6, Dr7;
    UINT64        Cr0, Cr1 /* Reserved */, Cr2, Cr3, Cr4, Cr8;
    UINT64        Rflags;
    UINT64        Ldtr, Tr;
    UINT64        Gdttr[2], Idtr[2];
    UINT64        Rip;
    UINT64        Gs, Fs, Es, Ds, Cs, Ss;
    UINT64        Rdi, Rsi, Rbp, Rsp, Rbx, Rdx, RcX, Rax;
    UINT64        R8, R9, R10, R11, R12, R13, R14, R15;
} EFI_SYSTEM_CONTEXT_X64;

// FXSAVE_STATE – FP / MMX / XMM registers
typedef struct {
    UINT16        Fcw;
    UINT16        Fsw;
    UINT16        Ftw;
    UINT16        Opcode;
    UINT64        Rip;
    UINT64        DataOffset;
    UINT8         Reserved1[8];
    UINT8         St0Mm0[10], Reserved2[6];
    UINT8         St1Mm1[10], Reserved3[6];
    UINT8         St2Mm2[10], Reserved4[6];
    UINT8         St3Mm3[10], Reserved5[6];
    UINT8         St4Mm4[10], Reserved6[6];
    UINT8         St5Mm5[10], Reserved7[6];
    UINT8         St6Mm6[10], Reserved8[6];
    UINT8         St7Mm7[10], Reserved9[6];
    UINT8         Xmm0[16];
    UINT8         Xmm1[16];
    UINT8         Xmm2[16];
    UINT8         Xmm3[16];
    UINT8         Xmm4[16];
    UINT8         Xmm5[16];
    UINT8         Xmm6[16];
    UINT8         Xmm7[16];
    UINT8         Reserved11[14 * 16];
} EFI_FX_SAVE_STATE_X64;
// System context for Itanium processor family
typedef struct {
    UINT64 Reserved;

    UINT64  R1, R2, R3, R4, R5, R6, R7, R8, R9, R10,
            R11, R12, R13, R14, R15, R16, R17, R18, R19, R20,
            R21, R22, R23, R24, R25, R26, R27, R28, R29, R30,
            R31;

    UINT64  F2[2], F3[2], F4[2], F5[2], F6[2],
            F7[2], F8[2], F9[2], F10[2], F11[2],
            F12[2], F13[2], F14[2], F15[2], F16[2],
            F17[2], F18[2], F19[2], F20[2], F21[2],
            F22[2], F23[2], F24[2], F25[2], F26[2],
            F27[2], F28[2], F29[2], F30[2], F31[2];

    UINT64  Pr;

    UINT64  B0, B1, B2, B3, B4, B5, B6, B7;

    // application registers
    UINT64  ArRsc, ArBsp, ArBspstore, ArRnat;
    UINT64  ArFcr;
    UINT64  ArEflag, ArCsd, ArSsd, ArCflg;
    UINT64  ArFsr, ArFir, ArFdr;
    UINT64  ArCcv;
    UINT64  ArUnat;
    UINT64  ArFpsr;
    UINT64  ArPfs, ArLc, ArEc;

    // control registers
    UINT64  CrDcr, CrItm, CrIva, CrPta, CrIpsr, CrIsr;
    UINT64  CrIip, CrIfa, CrItir, CrIipa, CrIfs, CrIim;
    UINT64  CrIha;

    // debug registers
    UINT64  Dbr0, Dbr1, Dbr2, Dbr3, Dbr4, Dbr5, Dbr6, Dbr7;
    UINT64  Ibr0, Ibr1, Ibr2, Ibr3, Ibr4, Ibr5, Ibr6, Ibr7;

    // virtual registers
    UINT64  IntNat;  // nat bits for R1-R31

} EFI_SYSTEM_CONTEXT_IPF;
/// ARM processor context definition
///
typedef struct {
  UINT32 R0;
  UINT32 R1;
  UINT32 R2;
  UINT32 R3;
  UINT32 R4;
  UINT32 R5;
  UINT32 R6;
  UINT32 R7;
  UINT32 R8;
  UINT32 R9;
  UINT32 R10;
  UINT32 R11;
  UINT32 R12;
  UINT32 SP;
  UINT32 LR;
  UINT32 PC;
  UINT32 CPSR;
  UINT32 DFSR;
  UINT32 DFAR;
  UINT32 IFSR;
} EFI_SYSTEM_CONTEXT_ARM;

/// AARCH64 processor context definition.
///
typedef struct {
  // General Purpose Registers
  UINT64 X0;
  UINT64 X1;
  UINT64 X2;
  UINT64 X3;
  UINT64 X4;
  UINT64 X5;
  UINT64 X6;
  UINT64 X7;
  UINT64 X8;
  UINT64 X9;
  UINT64 X10;
  UINT64 X11;
  UINT64 X12;
  UINT64 X13;
UINT64 X14;
UINT64 X15;
UINT64 X16;
UINT64 X17;
UINT64 X18;
UINT64 X19;
UINT64 X20;
UINT64 X21;
UINT64 X22;
UINT64 X23;
UINT64 X24;
UINT64 X25;
UINT64 X26;
UINT64 X27;
UINT64 X28;
UINT64 FP; // x29 - Frame Pointer
UINT64 LR; // x30 - Link Register
UINT64 SP; // x31 - Stack Pointer
    // FP/SIMD Registers
UINT64 V0[2];
UINT64 V1[2];
UINT64 V2[2];
UINT64 V3[2];
UINT64 V4[2];
UINT64 V5[2];
UINT64 V6[2];
UINT64 V7[2];
UINT64 V8[2];
UINT64 V9[2];
UINT64 V10[2];
UINT64 V11[2];
UINT64 V12[2];
UINT64 V13[2];
UINT64 V14[2];
UINT64 V15[2];
UINT64 V16[2];
UINT64 V17[2];
UINT64 V18[2];
UINT64 V19[2];
UINT64 V20[2];
UINT64 V21[2];
UINT64 V22[2];
UINT64 V23[2];
UINT64 V24[2];
UINT64 V25[2];
UINT64 V26[2];
UINT64 V27[2];
UINT64 V28[2];
UINT64 V29[2];
UINT64 V30[2];
UINT64 V31[2];
UINT64 ELR;    // Exception Link Register
UINT64 SPSR;   // Saved Processor Status Register
UINT64 FPSR;   // Floating Point Status Register
UINT64 ESR;    // Exception syndrome register
UINT64 FAR;    // Fault Address Register
} EFI_SYSTEM_CONTEXT_AARCH64;

Description

The `RegisterPeriodicCallback()` function registers and enables the on-target debug agent’s periodic entry point. To unregister and disable calling the debug agent’s periodic entry point, call `RegisterPeriodicCallback()` passing a `NULL` `PeriodicCallback` parameter.

The implementation must handle saving and restoring the processor context to/from the system context record around calls to the registered callback function.

If the interrupt is also used by the firmware for the EFI time base or some other use, two rules must be observed. First, the registered callback function must be called before any EFI processing takes place. Second, the Debug Support implementation must perform the necessary steps to pass control to the firmware’s corresponding interrupt handler in a transparent manner.

There is no quality of service requirement or specification regarding the frequency of calls to the registered `PeriodicCallback` function. This allows the implementation to mitigate a potential adverse impact to EFI timer based services due to the latency induced by the context save/restore and the associated callback function.

It is the responsibility of the caller to insure all parameters are correct. There is no provision for parameter checking by `RegisterPeriodicCallback()`. The implementation behavior when an invalid parameter is passed is not defined by this specification.

**Status Codes Returned**

<table>
<thead>
<tr>
<th>Status Code</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>EFI_SUCCESS</td>
<td>The function completed successfully.</td>
</tr>
<tr>
<td>EFI_ALREADY_STARTED</td>
<td>Non-NULL <code>PeriodicCallback</code> parameter when a callback function was previously registered.</td>
</tr>
<tr>
<td>EFI_OUT_OF_RESOURCES</td>
<td>System has insufficient memory resources to register new callback function.</td>
</tr>
</tbody>
</table>

**EFI_DEBUG_SUPPORT_PROTOCOL.RegisterExceptionCallback()**

Summary

Registers a function to be called when a given processor exception occurs.
Prototype

```c
typedef EFI_STATUS
(EFIAPI *REGISTER_EXCEPTION_CALLBACK) (
    IN EFI_DEBUG_SUPPORT_PROTOCOL *This,
    IN UINTN ProcessorIndex,
    IN EFI_EXCEPTION_CALLBACK ExceptionCallback,
    IN EFI_EXCEPTION_TYPE ExceptionType
);
```

Parameters

- **This**: A pointer to the `EFI_DEBUG_SUPPORT_PROTOCOL` instance. Type `EFI_DEBUG_SUPPORT_PROTOCOL` is defined in [Section 18.2](#).
- **ProcessorIndex**: Specifies which processor the callback function applies to.
- **ExceptionCallback**: A pointer to a function of type `EXCEPTION_CALLBACK` that is called when the processor exception specified by `ExceptionType` occurs. Passing `NULL` unregisters any previously registered function associated with `ExceptionType`.
- **ExceptionType**: Specifies which processor exception to hook.
typedef VOID (*EFI_EXCEPTION_CALLBACK) (  
    IN EFI_EXCEPTION_TYPE ExceptionType,  
    IN OUT EFI_SYSTEM_CONTEXT SystemContext  
);  

typedef INTN EFI_EXCEPTION_TYPE;  

// EBC Exception types  
#define EXCEPT_EBC_UNDEFINED                0  
#define EXCEPT_EBC_DIVIDE_ERROR             1  
#define EXCEPT_EBC_DEBUG                    2  
#define EXCEPT_EBC_BREAKPOINT               3  
#define EXCEPT_EBC_OVERFLOW                 4  
#define EXCEPT_EBC_INVALID_OPCODE           5  
#define EXCEPT_EBC_STACK_FAULT              6  
#define EXCEPT_EBC_ALIGNMENT_CHECK          7  
#define EXCEPT_EBC_INSTRUCTION_ENCODING     8  
#define EXCEPT_EBC_BAD_BREAK                9  
#define EXCEPT_EBC_SINGLE_STEP             10  

// IA-32 Exception types  
#define EXCEPT_IA32_DIVIDE_ERROR            0  
#define EXCEPT_IA32_DEBUG                   1  
#define EXCEPT_IA32_NMI                      2  
#define EXCEPT_IA32_BREAKPOINT              3  
#define EXCEPT_IA32_OVERFLOW                 4  
#define EXCEPT_IA32_BOUND                    5  
#define EXCEPT_IA32_INVALID_OPCODE           6  
#define EXCEPT_IA32_DOUBLE_FAULT            8  
#define EXCEPT_IA32_INVALID_TSS             10  
#define EXCEPT_IA32_SEG_NOT_PRESENT         11  
#define EXCEPT_IA32_STACK_FAULT              12  
#define EXCEPT_IA32_GP_FAULT                 13  
#define EXCEPT_IA32_PAGE_FAULT               14  
#define EXCEPT_IA32_FP_ERROR                 16  
#define EXCEPT_IA32_ALIGNMENT_CHECK          17  
#define EXCEPT_IA32_MACHINE_CHECK            18  
#define EXCEPT_IA32_SIMD                     19  

//  
// X64 Exception types  
//  
#define EXCEPT_X64_DIVIDE_ERROR             0
#define EXCEPT_X64_DEBUG                    1
#define EXCEPT_X64_NMI                      2
#define EXCEPT_X64_BREAKPOINT               3
#define EXCEPT_X64_OVERFLOW                 4
#define EXCEPT_X64_BOUND                    5
#define EXCEPT_X64_INVALID_OPCODE           6
#define EXCEPT_X64_DOUBLE_FAULT             8
#define EXCEPT_X64_INVALID_TSS              10
#define EXCEPT_X64_SEG_NOT_PRESENT          11
#define EXCEPT_X64_STACK_FAULT              12
#define EXCEPT_X64_GP_FAULT                 13
#define EXCEPT_X64_PAGE_FAULT               14
#define EXCEPT_X64_FP_ERROR                 16
#define EXCEPT_X64_ALIGNMENT_CHECK          17
#define EXCEPT_X64_MACHINE_CHECK            18
#define EXCEPT_X64_SIMD                     19

// Itanium Processor Family Exception types
#define EXCEPT_IPF_VHTP_TRANSLATION         0
#define EXCEPT_IPF_INSTRUCTION_TLB          1
#define EXCEPT_IPF_DATA_TLB                 2
#define EXCEPT_IPF_ALT_INSTRUCTION_TLB      3
#define EXCEPT_IPF_ALT_DATA_TLB             4
#define EXCEPT_IPF_DATA_NESTED_TLB          5
#define EXCEPT_IPF_INSTRUCTION_KEY_MISSED   6
#define EXCEPT_IPF_DATA_KEY_MISSED          7
#define EXCEPT_IPF_DIRTY_BIT                8
#define EXCEPT_IPF_INSTRUCTION_ACCESS_BIT   9
#define EXCEPT_IPF_DATA_ACCESS_BIT          10
#define EXCEPT_IPF_BREAKPOINT               11
#define EXCEPT_IPF_EXTERNAL_INTERRUPT       12
// 13 - 19 reserved
#define EXCEPT_IPF_PAGE_NOT_PRESENT         20
#define EXCEPT_IPF_KEY_PERMISSION           21
#define EXCEPT_IPF_INSTRUCTION_ACCESS_RIGHTS 22
#define EXCEPT_IPF_DATA_ACCESS_RIGHTS       23
#define EXCEPT_IPF_GENERAL_EXCEPTION        24
#define EXCEPT_IPF_DISABLED_FP_REGISTER     25
#define EXCEPT_IPF_NAT_CONSUMPTION          26
#define EXCEPT_IPF_SPECULATION              27
// 28 reserved
#define EXCEPT_IPF_DEBUG                    29
#define EXCEPT_IPF_UNALIGNED_REFERENCE      30
#define EXCEPT_IPF_UNSUPPORTED_DATA_REFERENCE 31
#define EXCEPT_IPF_FP_FAULT                 32
```c
#define EXCEPT_IPF_FP_TRAP 33
#define EXCEPT_IPF_LOWER_PRIVILEGE_TRANSFER_TRAP 34
#define EXCEPT_IPF_TAKEN_BRANCH 35
#define EXCEPT_IPF_SINGLE_STEP 36
// 37 - 44 reserved
#define EXCEPT_IPF_IA32_EXCEPTION 45
#define EXCEPT_IPF_IA32_INTERCEPT 46
#define EXCEPT_IPF_IA32_INTERRUPT 47

// ARM processor exception types
#define EXCEPT_ARM_RESET 0
#define EXCEPT_ARM_UNDEFINED_INSTRUCTION 1
#define EXCEPT_ARM_SOFTWARE_INTERRUPT 2
#define EXCEPT_ARM_PREFETCH_ABORT 3
#define EXCEPT_ARM_DATA_ABORT 4
#define EXCEPT_ARM_RESERVED 5
#define EXCEPT_ARM_IRQ 6
#define EXCEPT_ARM_FIQ 7

// For coding convenience, define the maximum valid ARM exception.
#define MAX_ARM_EXCEPTION EXCEPT_ARM_FIQ

// AARCH64 processor exception types.
#define EXCEPT_AARCH64_SYNCHRONOUS_EXCEPTIONS 0
#define EXCEPT_AARCH64_IRQ 1
#define EXCEPT_AARCH64_FIQ 2
#define EXCEPT_AARCH64_SERROR 3

// For coding convenience, define the maximum valid AARCH64 exception.
#define MAX_AARCH64_EXCEPTION EXCEPT_AARCH64_SERROR
```
///
/// RISC-V processor exception types.
///
#define EXCEPT_RISCV_INST_MISALIGNED              0
#define EXCEPT_RISCV_INST_ACCESS_FAULT            1
#define EXCEPT_RISCV_ILLEGAL_INST                 2
#define EXCEPT_RISCV_BREAKPOINT                   3
#define EXCEPT_RISCV_LOAD_ADDRESS_MISALIGNED      4
#define EXCEPT_RISCV_LOAD_ACCESS_FAULT            5
#define EXCEPT_RISCV_STORE_AMO_ADDRESS_MISALIGNED  6
#define EXCEPT_RISCV_STORE_AMO_ACCESS_FAULT       7
#define EXCEPT_RISCV_ENV_CALL_FROM_UMODE          8
#define EXCEPT_RISCV_ENV_CALL_FROM_SMODE          9
#define EXCEPT_RISCV_ENV_CALL_FROM_MMODE         11
#define EXCEPT_RISCV_INST_PAGE_FAULT              12
#define EXCEPT_RISCV_LOAD_PAGE_FAULT              13
#define EXCEPT_RISCV_STORE_AMO_PAGE_FAULT         15

///
/// RISC-V processor interrupt types.
///
#define EXCEPT_RISCV_SUPERVISOR_SOFTWARE_INT      1
#define EXCEPT_RISCV_MACHINE_SOFTWARE_INT         3
#define EXCEPT_RISCV_SUPERVISOR_TIMER_INT         5
#define EXCEPT_RISCV_MACHINE_TIMER_INT            7
#define EXCEPT_RISCV_SUPERVISOR_EXTERNAL_INT      9
#define EXCEPT_RISCV_MACHINE_EXTERNAL_INT        11

**Description**

The `RegisterExceptionCallback()` function registers and enables an exception callback function for the specified exception. The specified exception must be valid for the instruction set architecture. To unregister the callback function and stop servicing the exception, call `RegisterExceptionCallback()` passing a `NULL` `ExceptionCallback` parameter.

The implementation must handle saving and restoring the processor context to/from the system context record around calls to the registered callback function. No chaining of exception handlers is allowed.

It is the responsibility of the caller to insure all parameters are correct. There is no provision for parameter checking by `RegisterExceptionCallback()`. The implementation behavior when an invalid parameter is passed is not defined by this specification.
**Status Codes Returned**

<table>
<thead>
<tr>
<th>Code</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>EFI_SUCCESS</td>
<td>The function completed successfully.</td>
</tr>
<tr>
<td>EFI_ALREADY_STARTED</td>
<td>Non-NULL ExceptionCallback parameter when a callback function was previously registered.</td>
</tr>
<tr>
<td>EFI_OUT_OF_RESOURCES</td>
<td>System has insufficient memory resources to register new callback function.</td>
</tr>
</tbody>
</table>

**EFI_DEBUG_SUPPORT_PROTOCOL.InvalidateInstructionCache()**

**Summary**
Invalidates processor instruction cache for a memory range. Subsequent execution in this range causes a fresh memory fetch to retrieve code to be executed.

**Prototype**

```c
typedef EFI_STATUS (EFIAPI *EFI_INVALIDATE_INSTRUCTION_CACHE) (
    IN EFI_DEBUG_SUPPORT_PROTOCOL *This,
    IN UINTN ProcessorIndex,
    IN VOID *Start,
    IN UINT64 Length
);
```

**Parameters**

- **This**
  A pointer to the EFI_DEBUG_SUPPORT_PROTOCOL instance. Type EFI_DEBUG_SUPPORT_PROTOCOL is defined in Section 18.2.

- **ProcessorIndex**
  Specifies which processor’s instruction cache is to be invalidated.

- **Start**
  Specifies the physical base of the memory range to be invalidated.

- **Length**
  Specifies the minimum number of bytes in the processor’s instruction cache to invalidate.

**Description**

Typical operation of a debugger may require modifying the code image that is under debug. This can occur for many reasons, but is typically done to insert/remove software break instructions. Some processor architectures do not have coherent instruction and data caches so modifications to the code image require that the instruction cache be explicitly invalidated in that memory region.

The **InvalidateInstructionCache()** function abstracts this operation from the debug agent and provides a general purpose capability to invalidate the processor’s instruction cache.

It is the responsibility of the caller to insure all parameters are correct. There is no provision for parameter checking by EFI_DEBUG_SUPPORT_PROTOCOL.RegisterExceptionCallback(). The implementation behavior when an invalid parameter is passed is not defined by this specification.
### 18.3 EFI Debugport Protocol

This section defines the EFI Debugport protocol. This protocol is used by debug agent to communicate with the remote debug host.

#### 18.3.1 EFI Debugport Overview

Historically, remote debugging has typically been done using a standard UART serial port to connect the host and target. This is obviously not possible in a legacy reduced system that does not have a UART. The Debugport protocol solves this problem by providing an abstraction that can support many different types of debugport hardware. The debug agent should use this abstraction to communicate with the host.

The interface is minimal with only reset, read, write, and poll abstractions. Since these functions are called in interrupt context, none of them may call any EFI services or other protocol interfaces.

Debugport selection and configuration is handled by setting defaults via an environment variable which contains a full device path to the debug port. This environment variable is used during the debugport driver’s initialization to configure the debugport correctly. The variable contains a full device path to the debugport, with the last node (prior to the terminal node) being a debugport messaging node. See Section 18.3.2 for details.

The driver must also produce an instance of the EFI Device Path protocol to indicate what hardware is being used for the debugport. This may be used by the OS to maintain the debugport across a call to `EFI_BOOT_SERVICES.ExitBootServices()`.

**EFI_DEBUGPORT_PROTOCOL**

**Summary**

This protocol provides the communication link between the debug agent and the remote host.

**GUID**

```c
#define EFI_DEBUGPORT_PROTOCOL_GUID \
{0xEBA4E8D2,0x3858,0x41EC,\ 
  {0xA2,0x81,0x26,0x47,0xBA,0x96,0x60,0xD0}}
```
Protocol Interface Structure

typedef struct {
   EFI_DEBUGPORT_RESET   Reset;
   EFI_DEBUGPORT_WRITE   Write;
   EFI_DEBUGPORT_READ    Read;
   EFI_DEBUGPORT_POLL    Poll;
} EFI_DEBUGPORT_PROTOCOL;

Parameters
Reset
   Resets the debugport hardware.
Write
   Send a buffer of characters to the debugport device.
Read
   Receive a buffer of characters from the debugport device.
Poll
   Determine if there is any data available to be read from the debugport device.

Description
The Debugport protocol is used for byte stream communication with a debugport device. The debugport can be a standard UART Serial port, a USB-based character device, or potentially any character-based I/O device.

The attributes for all UART-style debugport device interfaces are defined in the DEBUGPORT variable (see Section 18.3.2).

EFI_DEBUGPORT_PROTOCOL.Reset()

Summary
Resets the debugport.

Prototype

typedef
   EFI_STATUS
   (EFIAPI *EFI_DEBUGPORT_RESET) (  
      IN EFI_DEBUGPORT_PROTOCOL   *This
   );

Parameters
This
   A pointer to the EFI_DEBUGPORT_PROTOCOL instance. Type EFI_DEBUGPORT_PROTOCOL is defined in Section 18.3.

Description
The Reset() function resets the debugport device.

It is the responsibility of the caller to insure all parameters are valid. There is no provision for parameter checking by Reset(). The implementation behavior when an invalid parameter is passed is not defined by this specification.
Status Codes Returned

<table>
<thead>
<tr>
<th>Code</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>EFI_SUCCESS</td>
<td>The debugport device was reset and is in usable state.</td>
</tr>
<tr>
<td>EFI_DEVICE_ERROR</td>
<td>The debugport device could not be reset and is unusable.</td>
</tr>
</tbody>
</table>

**EFI_DEBUGPORT_PROTOCOL.Write()**

**Summary**

Writes data to the debugport.

**Prototype**

```c
typedef EFI_STATUS
    (EFIAPI *EFI_DEBUGPORT_WRITE) (    
    IN EFI_DEBUGPORT_PROTOCOL  *This,    
    IN UINT32    Timeout,    
    IN OUT UINTN *BufferSize,    
    IN VOID       *Buffer    
    );
```

**Parameters**

- **This** A pointer to the EFI_DEBUGPORT_PROTOCOL instance. Type EFI_DEBUGPORT_PROTOCOL is defined in Section 18.3.
- **Timeout** The number of microseconds to wait before timing out a write operation.
- **BufferSize** On input, the requested number of bytes of data to write. On output, the number of bytes of data actually written.
- **Buffer** A pointer to a buffer containing the data to write.

**Description**

The `Write()` function writes the specified number of bytes to a debugport device. If a timeout error occurs while data is being sent to the debugport, transmission of this buffer will terminate, and EFI_TIMEOUT will be returned. In all cases the number of bytes actually written to the debugport device is returned in BufferSize.

It is the responsibility of the caller to insure all parameters are valid. There is no provision for parameter checking by `Write()`. The implementation behavior when an invalid parameter is passed is not defined by this specification.
Status Codes Returned

<table>
<thead>
<tr>
<th>Status Code</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>EFI_SUCCESS</td>
<td>The data was written.</td>
</tr>
<tr>
<td>EFI_DEVICE_ERROR</td>
<td>The device reported an error.</td>
</tr>
<tr>
<td>EFI_TIMEOUT</td>
<td>The data write was stopped due to a timeout.</td>
</tr>
</tbody>
</table>

**EFI_DEBUGPORT_PROTOCOL.Read()**

**Summary**
Reads data from the debugport.

**Prototype**
```c
typedef EFI_STATUS (EFIAPI *EFI_DEBUGPORT_READ) (  
    IN EFI_DEBUGPORT_PROTOCOL  *This,  
    IN UINT32                  Timeout,  
    IN OUT UINTN               *BufferSize,  
    OUT VOID                   *Buffer  
);
```

**Parameters**
- **This** A pointer to the EFI_DEBUGPORT_PROTOCOL instance. Type EFI_DEBUGPORT_PROTOCOL is defined in Section 18.3.
- **Timeout** The number of microseconds to wait before timing out a read operation.
- **BufferSize** A pointer to an integer which, on input contains the requested number of bytes of data to read, and on output contains the actual number of bytes of data read and returned in Buffer.
- **Buffer** A pointer to a buffer into which the data read will be saved.

**Description**
The Read() function reads a specified number of bytes from a debugport. If a timeout error or an overrun error is detected while data is being read from the debugport, then no more characters will be read, and EFI_TIMEOUT will be returned. In all cases the number of bytes actually read is returned in *BufferSize.

It is the responsibility of the caller to insure all parameters are valid. There is no provision for parameter checking by Read(). The implementation behavior when an invalid parameter is passed is not defined by this specification.
Status Codes Returned

<table>
<thead>
<tr>
<th>Status Code</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>EFI_SUCCESS</td>
<td>The data was read.</td>
</tr>
<tr>
<td>EFI_DEVICE_ERROR</td>
<td>The debugport device reported an error.</td>
</tr>
<tr>
<td>EFI_TIMEOUT</td>
<td>The operation was stopped due to a timeout or overrun.</td>
</tr>
</tbody>
</table>

EFI_DEBUGPORT_PROTOCOL.Poll()

Summary
Checks to see if any data is available to be read from the debugport device.

Prototype
```c
typedef EFI_STATUS (EFIAPI *EFI_DEBUGPORT_POLL) (
    IN EFI_DEBUGPORT_PROTOCOL *This
);
```

Parameters
- **This**: A pointer to the EFI_DEBUGPORT_PROTOCOL instance. Type EFI_DEBUGPORT_PROTOCOL is defined in Section 18.3.

Description
The Poll() function checks if there is any data available to be read from the debugport device and returns the result. No data is actually removed from the input stream. This function enables simpler debugger design since buffering of reads is not necessary by the caller.

Status Codes Returned

<table>
<thead>
<tr>
<th>Status Code</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>EFI_SUCCESS</td>
<td>At least one byte of data is available to be read.</td>
</tr>
<tr>
<td>EFI_NOT_READY</td>
<td>No data is available to be read.</td>
</tr>
<tr>
<td>EFI_DEVICE_ERROR</td>
<td>The debugport device is not functioning correctly.</td>
</tr>
</tbody>
</table>

18.3.2 Debugport Device Path

The debugport driver must establish and maintain an instance of the EFI Device Path protocol for the debugport. A graceful handoff of debugport ownership between the EFI Debugport driver and an OS debugport driver requires that the OS debugport driver can determine the type, location, and configuration of the debugport device.

The Debugport Device Path is a vendor-defined messaging device path with no data, only a GUID. It is used at the end of a conventional device path to tag the device for use as the debugport. For example, a typical UART debugport would have the following fully qualified device path:

```
PciRoot(0)/Pci(0x1f,0)/ACPI(PNP0501,0)/UART(115200,N,8,1)/DebugPort()
```

The Vendor_GUID that defines the debugport device path is the same as the debugport protocol GUID, as defined below.
Table 18-1 shows all fields of the debugport device path.

**Table 18-1 Debugport Messaging Device Path**

<table>
<thead>
<tr>
<th>Mnemonic</th>
<th>Byte Offset</th>
<th>Byte Length</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Type</td>
<td>0</td>
<td>1</td>
<td>Type 3 – Messaging Device Path.</td>
</tr>
<tr>
<td>Sub Type</td>
<td>1</td>
<td>1</td>
<td>Sub Type 10 – Vendor.</td>
</tr>
<tr>
<td>Length</td>
<td>2</td>
<td>2</td>
<td>Length of this structure in bytes. Length is 20 bytes.</td>
</tr>
<tr>
<td>Vendor_GUID</td>
<td>4</td>
<td>16</td>
<td>DEVICE_PATH_MESSAGING_DEBUGPORT.</td>
</tr>
</tbody>
</table>

### 18.3.3 EFI Debugport Variable

Even though there may be more than one hardware device that could function as a debugport in a system, only one debugport may be active at a time. The DEBUGPORT variable is used to declare which hardware device will act as the debugport, and what communication parameters it should assume.

Like all EFI variables, the DEBUGPORT variable has both a name and a GUID. The name is “DEBUGPORT.” The GUID is the same as the `EFI_DEBUGPORT_PROTOCOL_GUID`:

```c
#define EFI_DEBUGPORT_VARIABLE_NAME L"DEBUGPORT"
#define EFI_DEBUGPORT_VARIABLE_GUID EFI_DEBUGPORT_PROTOCOL_GUID
```

The data contained by the DEBUGPORT variable is a fully qualified debugport device path (see Section 18.3.2).

The desired communication parameters for the debugport are declared in the DEBUGPORT variable. The debugport driver must read this variable during initialization to determine how to configure the debug port.

To reduce the required complexity of the debugport driver, the debugport driver is not required to support all possible combinations of communication parameters. What combinations of parameters are possible is implementation specific.

Additionally debugport drivers implemented for PNP0501 devices, that is debugport devices with a PNP0501 ACPI node in the device path, must support the following defaults. These defaults must be used in the absence of a DEBUGPORT variable, or when the communication parameters specified in the DEBUGPORT variable are not supported by the driver.

- Baud : 115200
- 8 data bits
- No parity
- 1 stop bit
- No flow control (See Appendix A for flow control details)

In the absence of the DEBUGPORT variable, the selection of which port to use as the debug port is implementation specific.
Future revisions of this specification may define new defaults for other debugport types.

The debugport device path must be constructed to reflect the actual settings for the debugport. Any code needing to know the state of the debug port must reference the device path rather than the DEBUGPORT variable, since the debugport may have assumed a default setting in spite of the existence of the DEBUGPORT variable.

If it is not possible to configure the debug port using either the settings declared in the DEBUGPORT variable or the default settings for the particular debugport type, the driver initialization must not install any protocol interfaces and must exit with an error.

### 18.4 EFI Debug Support Table

This chapter defines the EFI Debug Support Table which is used by the debug agent or an external debugger to determine loaded image information in a quiescent manner.

#### 18.4.1 Overview

Every executable image loaded in EFI is represented by an EFI handle populated with an instance of the EFI_LOADED_IMAGE_PROTOCOL protocol. This handle is known as an “image handle.” The associated Loaded Image protocol provides image information that is of interest to a source level debugger. Normal EFI executables can access this information by using EFI services to locate all instances of the Loaded Image protocol.

A debugger has two problems with this scenario. First, if it is an external hardware debugger, the location of the EFI system table is not known. Second, even if the location of the EFI system table is known, the services contained therein are generally unavailable to a debugger either because it is an on-target debugger that is running in interrupt context, or in the case of an external hardware debugger there is no debugger code running on the target at all.

Since a source level debugger must be capable of determining image information for all loaded images, an alternate mechanism that does not use EFI services must be provided. Two features are added to the EFI system software to enable this capability.

First, an alternate mechanism of locating the EFI system table is required. A check-summed structure containing the physical address of the EFI system table is created and located on a 4M aligned memory address. A hardware debugger can search memory for this structure to determine the location of the EFI system table.

Second, an EFI_CONFIGURATION_TABLE is published that leads to a database of pointers to all instances of the Loaded Image protocol. Several layers of indirection are used to allow dynamically managing the data as images are loaded and unloaded. Once the address of the EFI system table is known, it is possible to discover a complete and accurate list of EFI images. (Note that the EFI core itself must be represented by an instance of the Loaded Image protocol.)

Figure 18-1 illustrates the table indirection and pointer usage.
18.4.2 EFI System Table Location

The EFI system table can be located by an off-target hardware debugger by searching for the EFI_SYSTEM_TABLE_POINTER structure. The EFI_SYSTEM_TABLE_POINTER structure is located on a 4M boundary as close to the top of physical memory as feasible. It may be found searching for the EFI_SYSTEM_TABLE_SIGNATURE on each 4M boundary starting at the top of memory and scanning down. When the signature is found, the entire structure must be verified using the Crc32 field. The 32-bit CRC of the entire structure is calculated assuming the Crc32 field is zero. This value is then written to the Crc32 field.

\[
\text{typedef struct _EFI_SYSTEM_TABLE_POINTER} \\
\quad \{ \\
\quad \quad \text{UINT64 Signature;} \\
\quad \quad \text{EFI_PHYSICAL_ADDRESS EfiSystemTableBase;} \\
\quad \quad \text{UINT32 Crc32;} \\
\quad \} \text{ EFI_SYSTEM_TABLE(Pointer);}
\]

**Signature**
A constant UINT64 that has the value EFI_SYSTEM_TABLE_SIGNATURE (see the EFI 1.0 specification).

**EfiSystemTableBase**
The physical address of the EFI system table.

**Crc32**
A 32-bit CRC value that is used to verify the EFI_SYSTEM_TABLE(Pointer) structure is valid.

18.4.3 EFI Image Info

The EFI_DEBUG_IMAGE_INFO_TABLE is an array of pointers to EFI_DEBUG_IMAGE_INFO unions. Each member of an EFI_DEBUG_IMAGE_INFO union is a pointer to a data structure representing a particular image type. For each image that has been loaded, there is an appropriate image data structure with a
pointer to it stored in the EFI_DEBUG_IMAGE_INFO_TABLE. Data structures for normal images and SMM images are defined. All other image types are reserved for future use.

The process of locating the EFI_DEBUG_IMAGE_INFO_TABLE begins with an EFI configuration table.

```
// EFI_DEBUG_IMAGE_INFO_TABLE configuration table
// GUID declaration - {49152E77-1ADA-4764-B7A2-7AFE0ED95E8B}

#define EFI_DEBUG_IMAGE_INFO_TABLE_GUID  
  {0x49152E77,0x1ADA,0x4764,\ 
   0xB7,0xA2,0x7A,0xFE,0xFE,0xD9,0x5E,0x8B }
```

The address reported in the EFI configuration table entry of this type will be referenced as physical and will not be fixed up when transition from preboot to runtime phase.

The configuration table leads to an EFI_DEBUG_IMAGE_INFO_TABLE_HEADER structure that contains a pointer to the EFI_DEBUG_IMAGE_INFO_TABLE and some status bits that are used to control access to the EFI_DEBUG_IMAGE_INFO_TABLE when it is being updated.

```
// UpdateStatus bits
//
#define EFI_DEBUG_IMAGE_INFO_UPDATE_IN_PROGRESS 0x01
#define EFI_DEBUG_IMAGE_INFO_TABLE_MODIFIED     0x02

typedef struct {
  volatile UINT32   UpdateStatus;
  UINT32           TableSize;
  EFI_DEBUG_IMAGE_INFO  *EfiDebugImageInfoTable;
} EFI_DEBUG_IMAGE_INFO_TABLE_HEADER;
```

**UpdateStatus**

UpdateStatus is used by the system to indicate the state of the debug image info table.

The EFI_DEBUG_IMAGE_INFO_UPDATE_IN_PROGRESS bit must be set when the table is being modified. Software consuming the table must qualify the access to the table with this bit.

The EFI_DEBUG_IMAGE_INFO_TABLE_MODIFIED bit is always set by software that modifies the table. It may be cleared by software that consumes the table once the entire table has been read. It is essentially a sticky version of the EFI_DEBUG_IMAGE_INFO_UPDATE_IN_PROGRESS bit and is intended to provide an efficient mechanism to minimize the number of times the table must be scanned by the consumer.

**TableSize**

The number of EFI_DEBUG_IMAGE_INFO elements in the array pointed to by EfiDebugImageInfoTable.

**EfiDebugImageInfoTable**

A pointer to the first element of an array of EFI_DEBUG_IMAGE_INFO structures.
#define EFI_DEBUG_IMAGE_INFO_TYPE_NORMAL 0x01

typedef union {
    UINT32 *ImageInfoType;
    EFI_DEBUG_IMAGE_INFO_NORMAL *NormalImage;
} EFI_DEBUG_IMAGE_INFO;

typedef struct {
    UINT32 ImageInfoType;
    EFI_LOADED_IMAGE_PROTOCOL *LoadedImageProtocolInstance;
    EFI_HANDLE ImageHandle;
} EFI_DEBUG_IMAGE_INFO_NORMAL;

ImageInfoType Indicates the type of image info structure. For PE32 EFI images, this is set to EFI_DEBUG_IMAGE_INFO_TYPE_NORMAL.

LoadedImageProtocolInstance A pointer to an instance of the loaded image protocol for the associated image.

ImageHandle Indicates the image handle of the associated image.
19 - Protocols — Compression Algorithm Specification

In EFI firmware storage, binary codes/data are often compressed to save storage space. These compressed codes/data are extracted into memory for execution at boot time. This demands an efficient lossless compression/decompression algorithm. The compressor must produce small compressed images, and the decompressor must operate fast enough to avoid delays at boot time.

This chapter describes in detail the UEFI compression/decompression algorithm, as well as the EFI Decompress Protocol. The EFI Decompress Protocol provides a standard decompression interface for use at boot time.

19.1 Algorithm Overview

In this chapter, the term “character” denotes a single byte and the term “string” denotes a series of concatenated characters.

The compression/decompression algorithm used in EFI firmware storage is a combination of the LZ77 algorithm and Huffman Coding. The LZ77 algorithm replaces a repeated string with a pointer to the previous occurrence of the string. Huffman Coding encodes symbols in a way that the more frequently a symbol appears in a text, the shorter the code that is assigned to it.

The compression process contains two steps:

- The first step is to find repeated strings (using LZ77 algorithm) and produce intermediate data.
  
  Beginning with the first character, the compressor scans the source data and determines if the characters starting at the current position can form a string previously appearing in the text. If a long enough matching string is found, the compressor will output a pointer to the string. If the pointer occupies more space than the string itself, the compressor will output the original character at the current position in the source data. Then the compressor advances to the next position and repeats the process. To speed up the compression process, the compressor dynamically maintains a String Info Log to record the positions and lengths of strings encountered, so that string comparisons are performed quickly by looking up the String Info Log.

  Because a compressor cannot have unlimited resources, as the compression continues the compressor removes “old” string information. This prevents the String Info Log from becoming too large. As a result, the algorithm can only look up repeated strings within the range of a fixed-sized “sliding window” behind the current position.

  In this way, a stream of intermediate data is produced which contains two types of symbols: the Original Characters (to be preserved in the decompressed data), and the Pointers (representing a previous string). A Pointer consists of two elements: the String Position and the String Length, representing the location and the length of the target string, respectively.

- To improve the compression ratio further, Huffman Coding is utilized as the second step.

  The intermediate data (consisting of original characters and pointers) is divided into Blocks so that the compressor can perform Huffman Coding on a Block immediately after it is generated; eliminating the need for a second pass from the beginning after the intermediate data has been generated. Also, since symbol frequency distribution may differ in different parts of the intermediate data, Huffman Coding can be optimized for each specific Block. The compressor determines Block Size for each Block according to the specifications defined in Section 19.2.
In each Block, two symbol sets are defined for Huffman Coding. The **Char&Len Set** consists of the Original Characters plus the String Lengths and the **Position Set** consists of String Positions (Note that the two elements of a Pointer belong to separate symbol sets). The Huffman Coding schemes applied on these two symbol sets are independent.

The algorithm uses “canonical” Huffman Coding so a Huffman tree can be represented as an array of code lengths in the order of the symbols in the symbol set. This code length array represents the Huffman Coding scheme for the symbol set. Both the Char&Len Set code length array and the Position Set code length array appear in the Block Header.

Huffman coding is used on the code length array of the Char&Len Set to define a third symbol set. The **Extra Set** is defined based on the code length values in the Char&Len Set code length array. The code length array for the Huffman Coding of Extra Set also appears in the Block Header together with the other two code length arrays. For exact format of the Block Header, see Section 19.2.3.1.

The decompression process is straightforward given that the compression process is known. The decompressor scans the compressed data and decodes the symbols one by one, according to the Huffman code mapping tables generated from code length arrays. Along the process, if it encounters an original character, it outputs it; if it encounters a pointer, it looks it up in the already decompressed data and outputs the associated string.

### 19.2 Data Format

This section describes in detail the format of the compressed data produced by the compressor. The compressed data serves as input to the decompressor and can be fully extracted to the original source data.

#### 19.2.1 Bit Order

In computer data representation, a byte is the minimum unit and there is no differentiation in the order of bits within a byte. However, the compressed data is a sequence of bits rather than a sequence of bytes and as a result the order of bits in a byte needs to be defined. In a compressed data stream, the higher bits are defined to precede the lower bits in a byte. **Figure 19-1** illustrates a compressed data sequence written as bytes from left to right. For each byte, the bits are written in an order with bit 7 (the highest bit) at the left and bit 0 (the lowest bit) at the right. Concatenating the bytes from left to right forms a bit sequence.

![Figure 19-1 Bit Sequence of Compressed Data](OM13173)
The bits of the compressed data are actually formed by a sequence of data units. These data units have variable bit lengths. The bits of each data unit are arranged so that the higher bit of the data unit precedes the lower bit of the data unit.

### 19.2.2 Overall Structure

The compressed data begins with two 32-bit numerical fields: the compressed size and the original size. The compressed data following these two fields is composed of one or more Blocks. Each Block is a unit for Huffman Coding with a coding scheme independent of the other Blocks. Each Block is composed of a Block Header containing the Huffman code trees for this Block and a Block Body with the data encoded using the coding scheme defined by the Huffman trees. The compressed data is terminated by an additional byte of zero.

The overall structure of the compressed data is shown in Figure 19-2.

![Figure 19-2 Compressed Data Structure](image)

Note the following:

- Blocks are of variable lengths.
- Block lengths are counted by bits and not necessarily divisible by 8. Blocks are tightly packed (there are no padding bits between blocks). Neither the starting position nor ending position of a Block is necessarily at a byte boundary. However, if the last Block is not terminated at a byte boundary, there should be some bits of 0 to fill up the remaining bits of the last byte of the block, before the terminator byte of 0.
- Compressed Size = Size in bytes of (Block 0 + Block 1 + ... + Block N + Filling Bits (if any) + Terminator).
- Original Size is the size in bytes of original data.
- Both Compressed Size and Original Size are “little endian” (starting from the least significant byte).

### 19.2.3 Block Structure

A Block is composed of a Block Header and a Block Body, as shown in Figure 19-3. These two parts are packed tightly (there are no padding bits between them). The lengths in bits of Block Header and Block Body are not necessarily divisible by eight.
19.2.3.1 Block Header

The Block Header contains the Huffman encoding information for this block. Since “canonical” Huffman Coding is being used, a Huffman tree is represented as an array of code lengths in increasing order of the symbols in the symbol set. Code lengths are limited to be less than or equal to 16 bits. This requires some extra handling of Huffman codes in the compressor, which is described in Section 19.3.

There are three code length arrays for three different symbol sets in the Block Header: one for the Extra Set, one for the Char&Len Set, and one for the Position Set.

The Block Header is composed of the tightly packed (no padding bits) fields described in Table 19-1.

Table 19-1 Block Header Fields

<table>
<thead>
<tr>
<th>Field Name</th>
<th>Length (bits)</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Block Size</td>
<td>16</td>
<td>The size of this Block. Block Size is defined as the number of original characters plus the number of pointers that appear in the Block Body: Block Size = Number of Original Characters in the Block Body + Number of Pointers in the Block Body.</td>
</tr>
<tr>
<td>Extra Set Code Length Array Size</td>
<td>5</td>
<td>The number of code lengths in the Extra Set Code Length Array. The Extra Set Code Length Array contains code lengths of the Extra Set in increasing order of the symbols, and if all symbols greater than a certain symbol have zero code length, the Extra Set Code Length Array terminates at the last nonzero code length symbol. Since there are 19 symbols in the Extra Set (see the description of the Char&amp;Len Set Code Length Array), the maximum Extra Set Code Length Array Size is 19.</td>
</tr>
<tr>
<td>Extra Set Code Length Array</td>
<td>Variable</td>
<td>If Extra Set Code Length Array Size is 0, then this field is a 5-bit value that represents the only Huffman code used. If Extra Set Code Length Array Size is not 0, then this field is an encoded form of a concatenation of code lengths in increasing order of the symbols. The concatenation of Code lengths are encoded as follows: If a code length is less than 7, then it is encoded as a 3-bit value; if a code length is equal to or greater than 7, then it is encoded as a series of “1”s followed by a terminating “0.” The number of “1”s = Code length – 4. For example, code length “ten” is encoded as “111110”; code length “seven” is encoded as “1110.” After the third length of the code length concatenation, a 2-bit value is used to indicate the number of consecutive zero lengths immediately after the third length. (Note this 2-bit value only appears once after the third length, and does NOT appear multiple times after every 3rd length.) This 2-bit value ranges from 0 to 3. For example, if the 2-bit value is “00,” then it means there are no zero lengths at the point, and following encoding starts from the fourth code length; if the 2-bit value is “10” then it means the fourth and fifth length are zero and following encoding starts from the sixth code length.</td>
</tr>
</tbody>
</table>
19.2.3.2 Block Body

The Block Body is simply a mixture of Original Characters and Pointers, while each Pointer has two elements: String Length preceding String Position. All these data units are tightly packed together.

<table>
<thead>
<tr>
<th>Field Name</th>
<th>Length (bits)</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Position Set Code Length Array Size</td>
<td>4</td>
<td>The number of code lengths in the Position Set Code Length Array. The Position Set Code Length Array contains code lengths of Position Set in increasing order of the symbols in the Position Set, and if all symbols greater than a certain symbol have zero code length, the Position Set Code Length Array terminates at the last nonzero code length symbol. Since there are 14 symbols in the Position Set (see 3.3.2), the maximum Position Set Code Length Array Size is 14.</td>
</tr>
<tr>
<td>Char&amp;Len Set Code Length Array</td>
<td>Variable</td>
<td>If Char&amp;Len Set Code Length Array Size is 0, then this field is a 9-bit value that represents the only Huffman code used. If Char&amp;Len Set Code Length Array Size is not 0, then this field is an encoded form of a concatenation of code lengths in increasing order of the symbols. The concatenation of Code lengths are two-step encoded: Step 1: If a code length is not zero, then it is encoded as “code length + 2”; if a code length is zero, then the number of consecutive zero lengths starting from this code length is counted -- if the count is equal to or less than 2, then the code “0” is used for each zero length; if the count is greater than 2 and less than 19, then the code “1” followed by a 4-bit value of “count – 3” is used for these consecutive zero lengths; if the count is equal to 19, then it is treated as “1 + 18,” and a code “0” and a code “1” followed by a 4-bit value of “15” are used for these consecutive zero lengths; if the count is greater than 19, then the code “2” followed by a 9-bit value of “count – 20” is used for these consecutive zero lengths. Step 2: The second step encoding is a Huffman encoding of the codes produced by first step. (While encoding codes “1” and “2,” their appended values are not encoded and preserved in the resulting text). The code lengths of generated Huffman tree are just the contents of the Extra Set Code Length Array.</td>
</tr>
<tr>
<td>Position Set Code Length Array Size</td>
<td>4</td>
<td>The number of code lengths in the Position Set Code Length Array. The Position Set Code Length Array contains code lengths of Position Set in increasing order of the symbols in the Position Set, and if all symbols greater than a certain symbol have zero code length, the Position Set Code Length Array terminates at the last nonzero code length symbol. Since there are 14 symbols in the Position Set (see 3.3.2), the maximum Position Set Code Length Array Size is 14.</td>
</tr>
<tr>
<td>Position Set Code Length Array</td>
<td>Variable</td>
<td>If Position Set Code Length Array Size is 0, then this field is a 5-bit value that represents the only Huffman code used. If Position Set Code Length Array Size is not 0, then this field is an encoded form of a concatenation of code lengths in increasing order of the symbols. The concatenation of Code lengths are encoded as follows: If a code length is less than 7, then it is encoded as a normal 3-bit value; if a code length is equal to or greater than 7, then it is encoded as a series of “1”s followed by a terminating “0.” The number of “1”s = Code length – 4. For example, code length “10” is encoded as “111110”; code length “7” is encoded as “1110.”</td>
</tr>
</tbody>
</table>
The Original Characters, String Lengths and String Positions are all Huffman coded using the Huffman trees presented in the Block Header, with some additional variations. The exact format is described below:

An Original Character is a byte in the source data. A String Length is a value that is greater than 3 and less than 257 (this range should be ensured by the compressor). By calculating “(String Length – 3) | 0x100,” a value set is obtained that ranges from 256 to 509. By combining this value set with the value set of Original Characters (0 ~ 255), the Char&Len Set (ranging from 0 to 509) is generated for Huffman Coding.

A String Position is a value that indicates the distance between the current position and the target string. The String Position value is defined as “Current Position – Starting Position of the target string - 1.” The String Position value ranges from 0 to 8190 (so 8192 is the “sliding window” size, and this range should be ensured by the compressor). The lengths of the String Position values (in binary form) form a value set ranging from 0 to 13 (it is assumed that value 0 has length of 0). This value set is the Position Set for Huffman Coding. The full representation of a String Position value is composed of two consecutive parts: one is the Huffman code for the value length; the other is the actual String Position value of “length - 1” bits (excluding the highest bit since the highest bit is always “1”). For example, String Position value 18 is represented as: Huffman code for “5” followed by “0010.” If the value length is 0 or 1, then no value is appended to the Huffman code. This kind of representation favors small String Position values, which is a hint for compressor design.

19.3 Compressor Design

The compressor takes the source data as input and produces a compressed image. This section describes the design used in one possible implementation of a compressor that follows the UEFI Compression Algorithm. The source code that illustrates an implementation of this specific design is listed in Appendix H.

19.3.1 Overall Process

The compressor scans the source data from the beginning, character by character. As the scanning proceeds, the compressor generates Original Characters or Pointers and outputs the compressed data packed in a series of Blocks representing individual Huffman coding units.

The compressor maintains a String Info Log containing data that facilitates string comparison. Old data items are deleted and new data items are inserted regularly.

The compressor does not output a Pointer immediately after it sees a matching string for the current position. Instead, it delays its decision until it gets the matching string for the next position. The compressor has two criteria at hand: one is that the former match length should be no shorter than three characters; the other is that the former match length should be no shorter than the latter match length.
Only when these two criteria are met does the compressor output a Pointer to the former matching string.

The overall process of compression can be described by following pseudo code:

```
Set the Current Position at the beginning of the source data;
Delete the outdated string info from the String Info Log;
Search the String Info Log for matching string;
Add the string info of the current position into the String Info Log;
WHILE not end of source data DO
    Remember the last match;
    Advance the Current Position by 1;
    Delete the outdated String Info from the String Info Log;
    Search the String Info Log for matching string;
    Add the string info of the Current Position into the String Info Log;
    IF the last match is shorter than 3 characters or this match is longer than
    the last match THEN
        Call Output() to output the character at the previous position as an
        Original Character;
    ELSE
        Call Output() to output a Pointer to the last matching string;
        WHILE (--last match length) > 0 DO
            Advance the Current Position by 1;
            Delete the outdated piece of string info from the String Info Log;
            Add the string info of the current position into the String Info Log;
        ENDWHILE
    ENDIF
ENDWHILE

The Output() is the function that is responsible for generating Huffman codes and Blocks. It accepts an Original Character or a Pointer as input and maintains a Block Buffer to temporarily store data units that are to be Huffman coded. The following pseudo code describes the function:

```
FUNCTION NAME: Output
INPUT: an Original Character or a Pointer

Put the Original Character or the Pointer into the Block Buffer;
Advance the Block Buffer position pointer by 1;
IF the Block Buffer is full THEN
    Encode the Char&Len Set in the Block buffer;
    Encode the Position Set in the Block buffer;
    Encode the Extra Set;
    Output the Block Header containing the code length arrays;
    Output the Block Body containing the Huffman encoded Original Characters and
    Pointers;
    Reset the Block Buffer position pointer to point to the beginning of the
    Block buffer;
ENDIF
```

19.3.2 String Info Log

The provision of the String Info Log is to speed up the process of finding matching strings. The design of this has significant impact on the overall performance of the compressor. This section describes in detail how String Info Log is implemented and the typical operations on it.
19.3.2.1 Data Structures

The String Info Log is implemented as a set of search trees. These search trees are dynamically updated as the compression proceeds through the source data. The structure of a typical search tree is depicted in Figure 19-5.

There are three types of nodes in a search tree: the root node, internal nodes, and leaves. The root node has a "character" attribute, which represents the starting character of a string. Each edge also has a "character" attribute, which represents the next character in the string. Each internal node has a "level" attribute, which indicates the character on any edge that leads to its child nodes is the "level + 1"th character in the string. Each internal node or leaf has a "position" attribute that indicates the string’s starting position in the source data.

To speed up the tree searching, a hash function is used. Given the parent node and the edge-character, the hash function will quickly find the expected child node.

19.3.2.2 Searching the Tree

Traversing the search tree is performed as follows:

The following example uses the search tree shown in Figure 19-5 above. Assume that the current position in the source data contains the string “camxrsxpj....”

1. The starting character “c” is used to find the root of the tree. The next character “a” is used to follow the edge from node 1 to node 2. The “position” of node 2 is 500, so a string starting with
“ca” can be found at position 500. The string at the current position is compared with the string starting at position 500.

2. Node 2 is at Level 3; so at most three characters are compared. Assume that the three-character comparison passes.

3. The fourth character “x” is used to follow the edge from Node 2 to Node 5. The position value of node 5 is 400, which means there is a string located in position 400 that starts with “cam” and the character at position 403 is an “x.”

4. Node 5 is at Level 8, so the fifth to eighth characters of the source data are compared with the string starting at position 404. Assume the strings match.

5. At this point, the ninth character “p” has been reached. It is used to follow the edge from Node 5 to Node 7.

6. This process continues until a mismatch occurs, or the length of the matching strings exceeds the predefined MAX_MATCH_LENGTH. The most recent matching string (which is also the longest) is the desired matching string.

19.3.2.3 Adding String Info

String info needs to be added to the String Info Log for each position in the source data. Each time a search for a matching string is performed, the new string info is inserted for the current position. There are several cases that can be discussed:

1. No root is found for the first character. A new tree is created with the root node labeled with the starting character and a child leaf node with its edge to the root node labeled with the second character in the string. The “position” value of the child node is set to the current position.

2. One root node matches the first character, but the second character does not match any edge extending from the root node. A new child leaf node is created with its edge labeled with the second character. The “position” value of the new leaf child node is set to the current position.

3. A string comparison succeeds with an internal node, but a matching edge for the next character does not exist. This is similar to (2) above. A new child leaf node is created with its edge labeled with the character that does not exist. The “position” value of the new leaf child node is set to the current position.

4. A string comparison exceeds MAX_MATCH_LENGTH. Note: This only happens with leaf nodes. For this case, the “position” value in the leaf node is updated with the current position.

5. If a string comparison with an internal node or leaf node fails (mismatch occurs before the “Level + 1”th character is reached or MAX_MATCH_LENGTH is exceeded), then a “split” operation is performed as follows:

Suppose a comparison is being performed with a level 9 Node, at position 350, and the current position is 1005. If the sixth character at position 350 is an “x” and the sixth character at position 1005 is a “y,” then a mismatch will occur. In this case, a new internal node and a new child node are inserted into the tree, as depicted in Figure 19-6.
Figure 19-6 Node Split

The b) portion of Figure 19-6 has two new inserted nodes, which reflects the new string information that was found at the current position. The process splits the old node into two child nodes, and that is why this operation is called a “split.”

19.3.2.4 Deleting String Info

The String Info Log will grow as more and more string information is logged. The size of the String Info Log must be limited, so outdated information must be removed on a regular basis. A sliding window is maintained behind the current position, and the searches are always limited within the range of the sliding window. Each time the current position is advanced, outdated string information that falls outside the sliding window should be removed from the tree. The search for outdated string information is simplified by always updating the nodes’ “position” attribute when searching for matching strings.

19.3.3 Huffman Code Generation

Another major component of the compressor design is generation of the Huffman Code.

Huffman Coding is applied to the Char&Len Set, the Position Set, and the Extra Set. The Huffman Coding used here has the following features:

- The Huffman tree is represented as an array of code lengths (“canonical” Huffman Coding);
- The maximum code length is limited to 16 bits.
The Huffman code generation process can be divided into three steps. These are the generation of Huffman tree, the adjustment of code lengths, and the code generation.

### 19.3.3.1 Huffman Tree Generation

This process generates a typical Huffman tree. First, the frequency of each symbol is counted, and a list of nodes is generated with each node containing a symbol and the symbol’s frequency. The two nodes with the lowest frequency values are merged into a single node. This new node becomes the parent node of the two nodes that are merged. The frequency value of this new parent node is the sum of the two child nodes’ frequency values. The node list is updated to include the new parent node but exclude the two child nodes that are merged. This process is repeated until there is a single node remaining that is the root of the generated tree.

### 19.3.3.2 Code Length Adjustment

The leaf nodes of the tree generated by the previous step represent all the symbols that were generated. Traditionally the code for each symbol is found by traversing the tree from the root node to the leaf node. Going down a left edge generates a “0,” and going down a right edge generates a “1.” However, a different approach is used here. The number of codes of each code length is counted. This generates a 16-element `LengthCount` array, with `LengthCount[i] = Number Of Codes whose Code Length is i`. Since a code length may be longer than 16 bits, the sixteenth entry of the `LengthCount` array is set to the Number Of Codes whose Code Length is greater than or equal to 16.

The `LengthCount` array goes through further adjustment described by following code:

```c
INT32 i, k;
UINT32 cum;

cum = 0;
for (i = 16; i > 0; i--) {
    cum += LengthCount[i] << (16 - i);
}
while (cum != (1U << 16)) {
    LengthCount[16]--;
    for (i = 15; i > 0; i--) {
        if (LengthCount[i] != 0) {
            LengthCount[i]--;
            LengthCount[i+1] += 2;
            break;
        }
    }
    cum--;
}
```

### 19.3.3.3 Code Generation

In the previous step, the count of each length was obtained. Now, each symbol is going to be assigned a code. First, the length of the code for each symbol is determined. Naturally, the code lengths are assigned in such a way that shorter codes are assigned to more frequently appearing symbols. A `CodeLength` array is generated with `CodeLength[i] = the code length of symbol i`. Given this array, a code
is assigned to each symbol using the algorithm described by the pseudo code below (the resulting codes are stored in array Code such that Code[i] = the code assigned to symbol i):

```c
INT32 i;
UINT16 Start[18];

Start[1] = 0;
for (i = 1; i <= 16; i++) {
    Start[i + 1] = (UINT16)((Start[i] + LengthCount[i]) << 1);
}
for (i = 0; i < NumberOfSymbols; i++) {
    Code[i] = Start[CodeLength[i]]++;  
}
```

The code length adjustment process ensures that no code longer than the designated length will be generated. As long as the decompressor has the CodeLength array at hand, it can regenerate the codes.

### 19.4 Decompressor Design

The decompressor takes the compressed data as input and produces the original source data. The main tasks for the decompressor are decoding Huffman codes and restoring Pointers to the strings to which they point.

The following pseudo code describes the algorithm used in the design of a decompressor. The source code that illustrates an implementation of this design is listed in Appendix I.

```
WHILE not end of data DO
    IF at block boundary THEN
        Read in the Extra Set Code Length Array;
        Generate the Huffman code mapping table for the Extra Set;
        Read in and decode the Char&Len Set Code Length Array;
        Generate the Huffman code mapping table for the Char&Len Set;
        Read in the Position Set Code Length Array;
        Generate the Huffman code mapping table for the Position Set;
        ENDF
        Get next code;
        Look the code up in the Char&Len Set code mapping table.
        Store the result as C;
        IF C < 256 (it represents an Original Character) THEN
            Output this character;
        ELSE (it represents a String Length)
            Transform C to be the actual String Length value;
            Get next code and look it up in the Position Set code mapping table, and with some additional transformation, store the result as P;
            Output C characters starting from the position “Current Position – P”;
        ENDIF
        ENDF
    ENDWHILE
```
19.5 Decompress Protocol

This section provides a detailed description of the `EFI_DECOMPRESS_PROTOCOL`.

**EFI_DECOMPRESS_PROTOCOL**

Summary

Provides a decompression service.

GUID

```c
#define EFI_DECOMPRESS_PROTOCOL_GUID \
  {0xd8117cfe,0x94a6,0x11d4,\ 
   {0x9a,0x3a,0x00,0x90,0x27,0x3f,0xc1,0x4d}}
```

Protocol Interface Structure

```c
typedef struct _EFI_DECOMPRESS_PROTOCOL {
  EFI_DECOMPRESS_GET_INFO   GetInfo;
  EFI_DECOMPRESS_DECOMPRESS Decompress;
} EFI_DECOMPRESS_PROTOCOL;
```

Parameters

**GetInfo**

Given the compressed source buffer, this function retrieves the size of the uncompressed destination buffer and the size of the scratch buffer required to perform the decompression. It is the caller’s responsibility to allocate the destination buffer and the scratch buffer prior to calling `EFI_DECOMPRESS_PROTOCOL.Decompress()`. See the `EFI_DECOMPRESS_PROTOCOL.GetInfo()` function description.

**Decompress**

Decompresses a compressed source buffer into an uncompressed destination buffer. It is the caller’s responsibility to allocate the destination buffer and a scratch buffer prior to making this call. See the `Decompress()` function description.

Description

The `EFI_DECOMPRESS_PROTOCOL` provides a decompression service that allows a compressed source buffer in memory to be decompressed into a destination buffer in memory. It also requires a temporary scratch buffer to perform the decompression. The `GetInfo()` function retrieves the size of the destination buffer and the size of the scratch buffer that the caller is required to allocate. The `Decompress()` function performs the decompression. The scratch buffer can be freed after the decompression is complete.

**EFI_DECOMPRESS_PROTOCOL.GetInfo()**

Summary

Given a compressed source buffer, this function retrieves the size of the uncompressed buffer and the size of the scratch buffer required to decompress the compressed source buffer.
Prototype

```c
typedef EFI_STATUS
(EFI_API *EFI_DECOMPRESS_GET_INFO) (
  IN  EFI_DECOMPRESS_PROTOCOL *This,
  IN  VOID *Source, 
  IN  UINT32 SourceSize, 
  OUT UINT32 *DestinationSize, 
  OUT UINT32 *ScratchSize 
);
```

Parameters

- **This**: A pointer to the EFI_DECOMPRESS_PROTOCOL instance. Type EFI_DECOMPRESS_PROTOCOL is defined in Section 19.5.
- **Source**: The source buffer containing the compressed data.
- **SourceSize**: The size, in bytes, of the source buffer.
- **DestinationSize**: A pointer to the size, in bytes, of the uncompressed buffer that will be generated when the compressed buffer specified by Source and SourceSize is decompressed.
- **ScratchSize**: A pointer to the size, in bytes, of the scratch buffer that is required to decompress the compressed buffer specified by Source and SourceSize.

Description

The `GetInfo()` function retrieves the size of the uncompressed buffer and the temporary scratch buffer required to decompress the buffer specified by Source and SourceSize. If the size of the uncompressed buffer or the size of the scratch buffer cannot be determined from the compressed data specified by Source and SourceData, then EFI_INVALID_PARAMETER is returned. Otherwise, the size of the uncompressed buffer is returned in DestinationSize, the size of the scratch buffer is returned in ScratchSize, and EFI_SUCCESS is returned.

The `GetInfo()` function does not have scratch buffer available to perform a thorough checking of the validity of the source data. It just retrieves the “Original Size” field from the beginning bytes of the source data and output it as DestinationSize. And ScratchSize is specific to the decompression implementation.
Status Codes Returned

<table>
<thead>
<tr>
<th>Status Code</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>EFI_SUCCESS</td>
<td>The size of the uncompressed data was returned in DestinationSize and the size of the scratch buffer was returned in ScratchSize.</td>
</tr>
<tr>
<td>EFI_INVALID_PARAMETER</td>
<td>The size of the uncompressed data or the size of the scratch buffer cannot be determined from the compressed data specified by Source and SourceSize.</td>
</tr>
</tbody>
</table>

**EFI_DECOMPRESS_PROTOCOL.Decompress()**

**Summary**

Decompresses a compressed source buffer.

**Prototype**

```c
typedef EFI_STATUS (EFIAPI *EFI_DECOMPRESS_DECOMPRESS) (
    IN  EFI_DECOMPRESS_PROTOCOL *This,
    IN   VOID      *Source,
    IN   UINT32    SourceSize,
    IN OUT VOID   *Destination,
    IN   UINT32    DestinationSize,
    IN OUT VOID   *Scratch,
    IN   UINT32    ScratchSize
);
```

**Parameters**

- **This**
  A pointer to the EFI_DECOMPRESS_PROTOCOL instance. Type **EFI_DECOMPRESS_PROTOCOL** is defined in [Section 19.5](#).

- **Source**
  The source buffer containing the compressed data.

- **SourceSize**
  The size of source data.

- **Destination**
  On output, the destination buffer that contains the uncompressed data.

- **DestinationSize**
  The size of the destination buffer. The size of the destination buffer needed is obtained from **EFI_DECOMPRESS_PROTOCOL.GetInfo()**.

- **Scratch**
  A temporary scratch buffer that is used to perform the decompression.

- **ScratchSize**
  The size of scratch buffer. The size of the scratch buffer needed is obtained from **GetInfo()**.

**Description**

The **Decompress()** function extracts decompressed data to its original form.
This protocol is designed so that the decompression algorithm can be implemented without using any memory services. As a result, the `Decompress()` function is not allowed to call `EFI_BOOT_SERVICES.AllocatePool()` or `EFI_BOOT_SERVICES.AllocatePages()` in its implementation. It is the caller’s responsibility to allocate and free the `Destination` and `Scratch` buffers.

If the compressed source data specified by `Source` and `SourceSize` is successfully decompressed into `Destination`, then `EFI_SUCCESS` is returned. If the compressed source data specified by `Source` and `SourceSize` is not in a valid compressed data format, then `EFI_INVALID_PARAMETER` is returned.

**Status Codes Returned**

<table>
<thead>
<tr>
<th>Status Code</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>EFI_SUCCESS</td>
<td>Decompression completed successfully, and the uncompressed buffer is returned in <code>Destination</code>.</td>
</tr>
<tr>
<td>EFI_INVALID_PARAMETER</td>
<td>The source buffer specified by <code>Source</code> and <code>SourceSize</code> is corrupted (not in a valid compressed format).</td>
</tr>
</tbody>
</table>
20 - Protocols — ACPI Protocols

EFI_ACPI_TABLE_PROTOCOL

Summary
This protocol may be used to install or remove an ACPI table from a platform.

GUID
#define EFI_ACPI_TABLE_PROTOCOL_GUID
{0xffe06bdd, 0x6107, 0x46a6,
{0x7b, 0xb2, 0x5a, 0x9c, 0x7e, 0xc5, 0x27, 0x5c}}

Protocol Interface Structure
typedef struct _EFI_ACPI_TABLE_PROTOCOL {
  EFI_ACPI_TABLE_INSTALL_ACPI_TABLE InstallAcpiTable;
  EFI_ACPI_TABLE_UNINSTALL_ACPI_TABLE UninstallAcpiTable;
} EFI_ACPI_TABLE_PROTOCOL;

Parameters
InstallAcpiTable Installs an ACPI table into the system.
UninstallAcpiTable Removes a previously installed ACPI table from the system.

Description
The EFI_ACPI_TABLE_PROTOCOL provides the ability for a component to install and uninstall ACPI tables from a platform.

EFI_ACPI_TABLE_PROTOCOL.InstallAcpiTable()

Summary
Installs an ACPI table into the RSDT/XSDT.

Prototype
typedef EFI_STATUS
(EIFIAPIC *EFI_ACPI_TABLE_INSTALL_ACPI_TABLE) (IN EFI_ACPI_TABLE_PROTOCOL *This,
IN VOID *AcpiTableBuffer,
IN UINTN AcpiTableBufferSize,
OUT UINTN *TableKey,
);

Parameters
This A pointer to a EFI_ACPI_TABLE_PROTOCOL.
AcpiTableBuffer A pointer to a buffer containing the ACPI table to be installed.
AcpiTableBufferSize
Specifies the size, in bytes, of the AcpiTableBuffer buffer.

TableKey
Returns a key to refer to the ACPI table.

Description
The InstallAcpiTable() function allows a caller to install an ACPI table. The ACPI table may either by a System Description Table or the FACS. For all tables except for the DSDT and FACS, a copy of the table will be linked by the RSDT/XSDT. For the FACS and DSDT, the pointer to a copy of the table will be updated in the FADT, if present.

To prevent namespace collision, ACPI tables may be created using UEFI ACPI table format. See Appendix O. If this protocol is used to install a table with a signature already present in the system, the new table will not replace the existing table. It is a platform implementation decision to add a new table with a signature matching an existing table or disallow duplicate table signatures and return EFI_ACCESS_DENIED.

On successful output, TableKey is initialized with a unique key. Its value may be used in a subsequent call to UninstallAcpiTable to remove an ACPI table.

On successful output, the EFI ACPI_TABLE_PROTOCOL will ensure that the checksum field is correct for both the RSDT/XSDT table and the copy of the table being installed that is linked by the RSDT/XSDT.

On successful completion, this function reinstalls the relevant EFI_CONFIGURATION_TABLE pointer to the RSDT.

Status Codes Returned

- EFI_SUCCESS
  The table was successfully inserted

- EFI_INVALID_PARAMETER
  The AcpiTableBuffer is NULL, the TableKey is NULL; the AcpiTableBufferSize, and the size field embedded in the ACPI table pointed to by AcpiTableBuffer are not in sync.

- EFI_OUT_OF_RESOURCES
  Insufficient resources exist to complete the request.

- EFI_ACCESS_DENIED
  The table signature matches a table already present in the system and platform policy does not allow duplicate tables of this type.

EFI ACPI_TABLE_PROTOCOL.UninstallAcpiTable()

Summary
Removes an ACPI table from the RSDT/XSDT.
Prototype
typedef EFI_STATUS (EFIAPI *EFI_ACPI_TABLE_UNINSTALL_ACPI_TABLE) ( 
    IN EFI_ACPI_TABLE_PROTOCOL *This, 
    IN UINTN TableKey, 
); 

Parameters
This A pointer to a EFI_ACPI_TABLE_PROTOCOL.
TableKey Specifies the table to uninstall. The key was returned from InstallAcpiTable().

Description
The UninstallAcpiTable() function allows a caller to remove an ACPI table. The routine will remove its reference from the RSDT/XSDT. A table is referenced by the TableKey parameter returned from a prior call to InstallAcpiTable().

On successful completion, this function reinstalls the relevant EFI_CONFIGURATION_TABLE pointer to the RSDT.

Status Codes Returned
  EFI_SUCCESS The table was successfully inserted
  EFI_NOT_FOUND TableKey does not refer to a valid key for a table entry.
  EFI_OUT_OF_RESOURCES Insufficient resources exist to complete the request.
21 - Protocols — String Services

21.1 Unicode Collation Protocol

This section defines the Unicode Collation protocol. This protocol is used to allow code running in the boot services environment to perform lexical comparison functions on Unicode strings for given languages.

EFI_UNICODE_COLLATION_PROTOCOL

Summary

Is used to perform case-insensitive comparisons of strings.

GUID

#define EFI_UNICODE_COLLATION_PROTOCOL2_GUID 
{
    0xa4c751fc, 0x23ae, 0x4c3e,
    {0x92, 0xe9, 0x49, 0x64, 0xcf, 0x63, 0xf3, 0x49}
}

Protocol Interface Structure

typedef struct {
    EFI_UNICODE_COLLATION_STMTTCOLL    StriColl;
    EFI_UNICODE_COLLATION_METAIMATCH    MetaiMatch;
    EFI_UNICODE_COLLATION_STRLWR        StrLwr;
    EFI_UNICODE_COLLATION_STRUPR        StrUp;
    EFI_UNICODE_COLLATION_FATTOSTR      FatToStr;
    EFI_UNICODE_COLLATION_STRTOFAT      StrToFat;
    CHAR8 *SupportedLanguages;
} EFI_UNICODE_COLLATION_PROTOCOL;

Parameters

StriColl        Performs a case-insensitive comparison of two Null-terminated strings. See the StriColl() function description.

MetaiMatch      Performs a case-insensitive comparison between a Null-terminated pattern string and a Null-terminated string. The pattern string can use the ‘?’ wildcard to match any character, and the ‘*’ wildcard to match any substring. See the MetaiMatch() function description.

StrLwr          Converts all the characters in a Null-terminated string to lowercase characters. See the StrLwr() function description.

StrUp            Converts all the characters in a Null-terminated string to uppercase characters. See the StrUp() function description.

FatToStr        Converts an 8.3 FAT file name using an OEM character set to a Null-terminated string. See the FatToStr() function description.

StrToFat        Converts a Null-terminated string to legal characters in a FAT filename using an OEM character set. See the StrToFat() function description.
SupportedLanguages

A Null-terminated ASCII string array that contains one or more language codes. This array is specified in RFC 4646 format. See Appendix M for the format of language codes and language code arrays.

Description

The EFI_UNICODE_COLLATION_PROTOCOL is used to perform case-insensitive comparisons of strings.

One or more of the EFI_UNICODE_COLLATION_PROTOCOL instances may be present at one time. Each protocol instance can support one or more language codes. The language codes supported in the EFI_UNICODE_COLLATION_PROTOCOL are declared in SupportedLanguages.

The SupportedLanguages is a Null-terminated ASCII string array that contains one or more supported language codes. This is the list of language codes that this protocol supports. See Appendix M for the format of language codes and language code arrays.

The main motivation for this protocol is to help support file names in a file system driver. When a file is opened, a file name needs to be compared to the file names on the disk. In some cases, this comparison needs to be performed in a case-insensitive manner. In addition, this protocol can be used to sort files from a directory or to perform a case-insensitive file search.

EFI_UNICODE_COLLATION_PROTOCOL.StriColl()

Summary

Performs a case-insensitive comparison of two Null-terminated strings.

Prototype

typedef INTN (EFIAPI *EFI_UNICODE_COLLATION_STRICOLL) (IN EFI_UNICODE_COLLATION_PROTOCOL *This,
IN CHAR16 *s1,
IN CHAR16 *s2);

Parameters

This A pointer to the EFI_UNICODE_COLLATION_PROTOCOL instance. Type EFI_UNICODE_COLLATION_PROTOCOL is defined above.

s1 A pointer to a Null-terminated string.

s2 A pointer to a Null-terminated string.

Description

The StriColl() function performs a case-insensitive comparison of two Null-terminated strings.

This function performs a case-insensitive comparison between the string $s1$ and the string $s2$ using the rules for the language codes that this protocol instance supports. If $s1$ is equivalent to $s2$, then 0 is returned. If $s1$ is lexically less than $s2$, then a negative number will be returned. If $s1$ is lexically greater than $s2$, then a positive number will be returned. This function allows strings to be compared and sorted.
**Status Codes Returned**

<table>
<thead>
<tr>
<th>Status Code</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>s1 is equivalent to s2.</td>
</tr>
<tr>
<td>&gt; 0</td>
<td>s1 is lexically greater than s2.</td>
</tr>
<tr>
<td>&lt; 0</td>
<td>s1 is lexically less than s2.</td>
</tr>
</tbody>
</table>

**EFI_UNICODE_COLLATION_PROTOCOL.MetaiMatch()**

**Summary**
Performs a case-insensitive comparison of a Null-terminated pattern string and a Null-terminated string.

**Prototype**

```c
typedef BOOLEAN (EFIAPI *EFI_UNICODE_COLLATION_METAIMATCH) (
    IN EFI_UNICODE_COLLATION_PROTOCOL *This,
    IN CHAR16 *String,
    IN CHAR16 *Pattern
);
```

**Parameters**

- **This**
  A pointer to the EFI_UNICODE_COLLATION_PROTOCOL instance. Type **EFI_UNICODE_COLLATION_PROTOCOL** is defined above.
- **String**
  A pointer to a Null-terminated string.
- **Pattern**
  A pointer to a Null-terminated string.

**Description**

The **MetaiMatch()** function performs a case-insensitive comparison of a Null-terminated pattern string and a Null-terminated string.

This function checks to see if the pattern of characters described by **Pattern** are found in **String**. The pattern check is a case-insensitive comparison using the rules for the language codes that this protocol instance supports. If the pattern match succeeds, then **TRUE** is returned. Otherwise **FALSE** is returned.

The following syntax can be used to build the string **Pattern**:

- *****
  Match 0 or more characters.
- **?**
  Match any one character.
- **[<char1><char2>...<charN>]**
  Match any character in the set.
- **[<char1>-<char2>]**
  Match any character between <char1> and <char2>.
- **<char>**
  Match the character <char>.

Following is an example pattern for English:

- ***.FW**
  Matches all strings that end in “.FW” or “.fw” or “.Fw” or “.FW.”
- **[a-z]**
  Match any letter in the alphabet.
- **[@#$%^&*()]**
  Match any one of these symbols.
**String Services**

### Status Codes Returned

<table>
<thead>
<tr>
<th>TRUE</th>
<th>Pattern was found in String.</th>
</tr>
</thead>
<tbody>
<tr>
<td>FALSE</td>
<td>Pattern was not found in String.</td>
</tr>
</tbody>
</table>

### EFI_UNICODE_COLLATION_PROTOCOL.StrLwr()

**Summary**

Converts all the characters in a Null-terminated string to lowercase characters.

**Prototype**

```c
typedef VOID (EFIAPI *EFI_UNICODE_COLLATION_StrLwr) (
    IN EFI_UNICODE_COLLATION_PROTOCOL *This,
    IN OUT CHAR16 *String
);
```

**Parameters**

- **This**
  
  A pointer to the EFI_UNICODE_COLLATION_PROTOCOL instance. Type **EFI_UNICODE_COLLATION_PROTOCOL** is defined above.

- **String**
  
  A pointer to a Null-terminated string.

**Description**

This function walks through all the characters in String, and converts each one to its lowercase equivalent if it has one. The converted string is returned in String.

### EFI_UNICODE_COLLATION_PROTOCOL.StrUpr()

**Summary**

Converts all the characters in a Null-terminated string to uppercase characters.
Prototype

typedef VOID
    (EFIAPI *EFI_UNICODE_COLLATION_STRUPR) (  
        IN EFI_UNICODE_COLLATION_PROTOCOL *This,  
        IN OUT CHAR16 *String
    );

Parameters

    This A pointer to the EFI_UNICODE_COLLATION_PROTOCOL instance.  
        Type EFI_UNICODE_COLLATION_PROTOCOL is defined above.

    String A pointer to a Null-terminated string.

Description

This function walks through all the characters in String, and converts each one to its uppercase 
equivalent if it has one. The converted string is returned in String.

EFI_UNICODE_COLLATION_PROTOCOL.FatToStr()

Summary

Converts an 8.3 FAT file name in an OEM character set to a Null-terminated string.

Prototype

    typedef VOID
        (EFIAPI *EFI_UNICODE_COLLATION_FATTOSTR) (  
            IN EFI_UNICODE_COLLATION_PROTOCOL *This,  
            IN UINTN FatSize,  
            IN CHAR8 *Fat,  
            OUT CHAR16 *String
        );

Parameters

    This A pointer to the EFI_UNICODE_COLLATION_PROTOCOL instance.  
        Type EFI_UNICODE_COLLATION_PROTOCOL is defined above.

    FatSize The size of the string Fat in bytes.

    Fat A pointer to a Null-terminated string that contains an 8.3 file name  
        encoded using an 8-bit OEM character set.

    String A pointer to a Null-terminated string. The string must be allocated in  
        advance to hold FatSize characters.

Description

This function converts the string specified by Fat with length FatSize to the Null-terminated string  
specified by String. The characters in Fat are from an OEM character set.
**EFI_UNICODE_COLLATION_PROTOCOL.StrToFat()**

**Summary**
Converts a Null-terminated string to legal characters in a FAT filename using an OEM character set.

**Prototype**
```c
typedef
BOOLEAN
(EIFIAPI *EFI_UNICODE_COLLATION_STRTOFAT) (  
   IN EFI_UNICODE_COLLATION_PROTOCOL *This,  
   IN CHAR16 *String,  
   IN UINTN FatSize,  
   OUT CHAR8 *Fat
);
```

**Parameters**
- **This**: A pointer to the EFI_UNICODE_COLLATION_PROTOCOL instance. Type EFI_UNICODE_COLLATION_PROTOCOL is defined above.
- **String**: A pointer to a Null-terminated string.
- **FatSize**: The size of the string Fat in bytes.
- **Fat**: A pointer to a string that contains the converted version of String using legal FAT characters from an OEM character set.

**Description**
This function converts the characters from String into legal FAT characters in an OEM character set and stores them in the string Fat. This conversion continues until either FatSize bytes are stored in Fat, or the end of String is reached. The characters ‘.’ (period) and ‘ ’ (space) are ignored for this conversion. Characters that map to an illegal FAT character are substituted with an ‘_’. If no valid mapping from a character to an OEM character is available, then it is also substituted with an ‘_’. If any of the character conversions are substituted with a ‘_’, then TRUE is returned. Otherwise FALSE is returned.

**Status Codes Returned**

<table>
<thead>
<tr>
<th>Status</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>TRUE</td>
<td>One or more conversions failed and were substituted with ‘_’.</td>
</tr>
<tr>
<td>FALSE</td>
<td>None of the conversions failed.</td>
</tr>
</tbody>
</table>

**21.2 Regular Expression Protocol**

This section defines the Regular Expression Protocol. This protocol is used to match Unicode strings against Regular Expression patterns.
**EFI_REGULAR_EXPRESSION_PROTOCOL**

**Summary**

**GUID**

```c
define EFI_REGULAR_EXPRESSION_PROTOCOL_GUID \ 
{ 0xB3F79D9A, 0x436C, 0xDC11,\ 
{ 0xB0, 0x52, 0xCD, 0x85, 0xDF, 0x52, 0x4C, 0xE6 } }
```

**Protocol Interface Structure**

```c
typedef struct {
    EFI_REGULAR_EXPRESSION_MATCH MatchString;
    EFI_REGULAR_EXPRESSION_GET_INFO GetInfo;
} EFI_REGULAR_EXPRESSION_PROTOCOL;
```

**Parameters**

- **MatchString**
  Search the input string for anything that matches the regular expression.

- **GetInfo**
  Returns information about the regular expression syntax types supported by the implementation.

**EFI_REGULAR_EXPRESSION_PROTOCOL.MatchString()**

**Summary**

Checks if the input string matches to the regular expression pattern.

**Prototype**

```c
typedef EFI_STATUS
EFI_API *EFI_REGULAR_EXPRESSION_MATCH) ( 
    IN   EFI_REGULAR_EXPRESSION_PROTOCOL *This,
    IN   CHAR16            *String,
    IN   CHAR16            *Pattern,
    IN   EFI_REGEX_SYNTAX_TYPE     *SyntaxType, OPTIONAL
    OUT  BOOLEAN            *Result,
    OUT  EFI_REGEX_CAPTURE       **Captures, OPTIONAL
    OUT UINTN              *CapturesCount
);
```

**Parameters**

- **This**
  A pointer to the `EFI_REGULAR_EXPRESSION_PROTOCOL` instance. Type `EFI_REGULAR_EXPRESSION_PROTOCOL` is defined in above.

- **String**
  A pointer to a NULL terminated string to match against the regular expression string specified by `Pattern`.

- **Pattern**
  A pointer to a NULL terminated string that represents the regular expression.
SyntaxType

A pointer to the `EFI_REGEX_SYNTAX_TYPE` that identifies the regular expression syntax type to use. May be NULL in which case the function will use its default regular expression syntax type.

Result

On return, points to `TRUE` if `String` fully matches against the regular expression `Pattern` using the regular expression `SyntaxType`. Otherwise, points to `FALSE`.

Captures

A Pointer to an array of `EFI_REGEX_CAPTURE` objects to receive the captured groups in the event of a match. The full sub-string match is put in `Captures[0]`, and the results of N capturing groups are put in `Captures[1:N]`. If `Captures` is NULL, then this function doesn't allocate the memory for the array and does not build up the elements. It only returns the number of matching patterns in `CapturesCount`. If `Captures` is not NULL, this function returns a pointer to an array and builds up the elements in the array. `CapturesCount` is also updated to the number of matching patterns found. It is the caller’s responsibility to free the memory pool in `Captures` and in each `CapturePtr` in the array elements.

CapturesCount

On output, `CapturesCount` is the number of matching patterns found in `String`. Zero means no matching patterns were found in the string.

Description

The `MatchString()` function performs a matching of a Null-terminated input string with the NULL-terminated pattern string. The pattern string syntax type is optionally identified in `SyntaxType`.

This function checks to see if `String` fully matches against the regular expression described by `Pattern`. The pattern check is performed using regular expression rules that are supported by this implementation, as indicated in the return value of `GetInfo` function. If the pattern match succeeds, then `TRUE` is returned in `Result`. Otherwise `FALSE` is returned.

Related Definitions

```c
typedef struct {
    CONST CHAR16 *CapturePtr;
    UINTN Length;
} EFI_REGEX_CAPTURE;
```

*CapturePtr

Pointer to the start of the captured sub-expression within matched `String`.

Length

Length of captured sub-expression.
Status Codes Returned

<table>
<thead>
<tr>
<th>Code</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>EFI_SUCCESS</td>
<td>The regular expression string matching completed successfully.</td>
</tr>
<tr>
<td>EFI_UNSUPPORTED</td>
<td>The regular expression syntax specified by SyntaxType is not supported by this</td>
</tr>
<tr>
<td></td>
<td>driver.</td>
</tr>
<tr>
<td>EFI_DEVICE_ERROR</td>
<td>The regular expression string matching failed due to a hardware or firmware</td>
</tr>
<tr>
<td></td>
<td>error.</td>
</tr>
<tr>
<td>EFI_INVALID_PARAMETER</td>
<td>String, Pattern, Result, or CapturesCount is NULL.</td>
</tr>
</tbody>
</table>

EFI_REGULAR_EXPRESSION_PROTOCOL.GetInfo()

Summary
Returns information about the regular expression syntax types supported by the implementation.

Prototype
```c
typedef
EFI_STATUS
EFIAPIC *EFI_REGULAR_EXPRESSION_GET_INFO) (  
    IN   EFI_REGULAR_EXPRESSION_PROTOCOL* This,    
    IN OUT UINTN             *RegExSyntaxTypeListSize,  
    OUT  EFI_REGEX_SYNTAX_TYPE *RegExSyntaxTypeList 
);
```

Parameters
- **This**: A pointer to the EFI_REGULAR_EXPRESSION_PROTOCOL instance.
- **RegExSyntaxTypeListSize**: On input, the size in bytes of RegExSyntaxTypeList. On output with a return code of EFI_SUCCESS, the size in bytes of the data returned in RegExSyntaxTypeList. On output with a return code of EFI_BUFFER_TOO_SMALL, the size of RegExSyntaxTypeList required to obtain the list.
- **RegExSyntaxTypeList**: A caller-allocated memory buffer filled by the driver with one EFI_REGEX_SYNTAX_TYPE element for each supported regular expression syntax type. The list must not change across multiple calls to the same driver. The first syntax type in the list is the default type for the driver.

Description
This function returns information about supported regular expression syntax types. A driver implementing the EFI_REGULAR_EXPRESSION_PROTOCOL protocol need not support more than one regular expression syntax type, but shall support a minimum of one regular expression syntax type.
Related Definitions

typedef EFI_GUID EFI_REGEX_SYNTAX_TYPE;

Status Codes Returned

<table>
<thead>
<tr>
<th>Status Code</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>EFI_SUCCESS</td>
<td>The regular expression syntax types list was returned successfully.</td>
</tr>
<tr>
<td>EFI_UNSUPPORTED</td>
<td>The service is not supported by this driver.</td>
</tr>
<tr>
<td>EFI_DEVICE_ERROR</td>
<td>The list of syntax types could not be retrieved due to a hardware or firmware error.</td>
</tr>
<tr>
<td>EFI_BUFFER_TOO_SMALL</td>
<td>The buffer RegExSyntaxTypeList is too small to hold the result.</td>
</tr>
<tr>
<td>EFI_INVALID_PARAMETER</td>
<td>RegExSyntaxTypeListSize is NULL.</td>
</tr>
</tbody>
</table>

21.2.1 EFI Regular Expression Syntax Type Definitions

Summary

This sub-section provides EFI_GUID values for a selection of EFI_REGULAR_EXPRESSION_PROTOCOL syntax types. The types listed are optional, not meant to be exhaustive and may be augmented by vendors or other industry standards.

Prototype

For regular expression rules specified in the POSIX Extended Regular Expression (ERE) Syntax:

```c
#define EFI_REGEX_SYNTAX_TYPE_POSIX_EXTENDED_GUID \ 
  {0x5F05B20F, 0x4A56, 0xC231,\ 
   { 0xFA, 0x0B, 0xA7, 0xB1, 0xF1, 0x10, 0x04, 0x1D }}
```

For regular expression rules specified in the Perl standard:

```c
#define EFI_REGEX_SYNTAX_TYPE_PERL_GUID \ 
  {0x63E60A51, 0x497D, 0xD427,\ 
   { 0xC4, 0xA5, 0xB8, 0xAB, 0xDC, 0x3A, 0xAE, 0xB6 }}
```

For regular expression rules specified in the ECMA 262 Specification:

```c
#define EFI_REGEX_SYNTAX_TYPE_ECMA_262_GUID \ 
  { 0x9A473A4A, 0x4CEB, 0xB95A, 0x41,\ 
    { 0x5E, 0x5B, 0xA0, 0xBC, 0x63, 0x9B, 0x2E }}
```

For regular expression rules specified in the POSIX Extended Regular Expression (ERE) Syntax, where the Pattern and String input strings need to be converted to ASCII:
#define EFI_REGEX_SYNTAX_TYPE_POSIX_EXTENDED_ASCII_GUID  
{0x3FD32128, 0x4BB1, 0xF632,  
 { 0xBE, 0x4F, 0xBA, 0xBF, 0x85, 0xC9, 0x36, 0x76 }}

For regular expression rules specified in the Perl standard, where the Pattern and String input strings need to be converted to ASCII:

#define EFI_REGEX_SYNTAX_TYPE_PERL_ASCII_GUID  
{0x87DFB76D, 0x4B58, 0xEF3A,  
 { 0xF7, 0xC6, 0x16, 0xA4, 0x2A, 0x68, 0x28, 0x10 }}

For regular expression rules specified in the ECMA 262 Specification, where the Pattern and String input strings need to be converted to ASCII:

#define EFI_REGEX_SYNTAX_TYPE_ECMA_262_ASCII_GUID  
{ 0xB2284A2F, 0x4491, 0x6D9D,  
 { 0xEA, 0xB7, 0x11, 0xB0, 0x67, 0xD4, 0x9B, 0x9A }}

(See Appendix Q for more information.)

(See References for more information.)
22 - EFI Byte Code Virtual Machine

This section defines an EFI Byte Code (EBC) Virtual Machine that can provide platform- and processor-independent mechanisms for loading and executing EFI device drivers.

22.1 Overview

The current design for option ROMs that are used in personal computer systems has been in place since 1981. Attempts to change the basic design requirements have failed for a variety of reasons. The EBC Virtual Machine described in this chapter is attempting to help achieve the following goals:

- Abstract and extensible design
- Processor independence
- OS independence
- Build upon existing specifications when possible
- Facilitate the removal of legacy infrastructure
- Exclusive use of EFI Services

One way to satisfy many of these goals is to define a pseudo or virtual machine that can interpret a predefined instruction set. This will allow the virtual machine to be ported across processor and system architectures without changing or recompiling the option ROM. This specification defines a set of machine level instructions that can be generated by a C compiler.

The following sections are a detailed description of the requirements placed on future option ROMs.

22.1.1 Processor Architecture Independence

Option ROM images shall be independent of supported 32-bit and supported 64-bit architectures. In order to abstract the architectural differences between processors option ROM images shall be EBC. This model is presented below:

- 64-bit C source code
- The EFI EBC image is the flashed image
- The system BIOS implements the EBC interpreter
- The interpreter handles 32 vs. 64 bit issues

Current Option ROM technology is processor dependent and heavily reliant upon the existence of the PC-AT infrastructure. These dependencies inhibit the evolution of both hardware and software under the veil of “backward compatibility.” A solution that isolates the hardware and support infrastructure through abstraction will facilitate the uninhibited progression of technology.

22.1.2 OS Independent

Option ROMs shall not require or assume the existence of a particular OS.

22.1.3 EFI Compliant

Option ROM compliance with EFI requires (but is not limited to) the following:
• Little endian layout
• Single-threaded model with interrupt polling if needed
• Where EFI provides required services, EFI is used exclusively. These include:
  — Console I/O
  — Memory Management
  — Timer services
  — Global variable access
• When an Option ROM provides EFI services, the EFI specification is strictly followed:
  — Service/protocol installation
  — Calling conventions
  — Data structure layouts
  — Guaranteed return on services

22.1.4 Coexistence of Legacy Option ROMs
The infrastructure shall support coexistent Legacy Option ROM and EBC Option ROM images. This case would occur, for example, when a Plug and Play Card has both Legacy and EBC Option ROM images flashed. The details of the mechanism used to select which image to load is beyond the scope of this document. Basically, a legacy System BIOS would not recognize an EBC Option ROM and therefore would never load it. Conversely, an EFI Firmware Boot Manager would only load images that it supports.

The EBC Option ROM format must utilize a legacy format to the extent that a Legacy System BIOS can:
• Determine the type of the image, in order to ignore the image. The type must be incompatible with currently defined types.
• Determine the size of the image, in order to skip to the next image.

22.1.5 Relocatable Image
An EBC option ROM image shall be eligible for placement in any system memory area large enough to accommodate it.

Current option ROM technology requires images to be shadowed in system memory address range 0xC0000 to 0xEFFFF on a 2048 byte boundary. This dependency not only limits the number of Option ROMs, it results in unused memory fragments up to 2 KiB.

22.1.6 Size Restrictions Based on Memory Available
EBC option ROM images shall not be limited to a predetermined fixed maximum size.

Current option ROM technology limits the size of a preinitialization option ROM image to 128 KiB (126 KiB actual). Additionally, in the DDIM an image is not allowed to grow during initialization. It is inevitable that 64-bit solutions will increase in complexity and size. To avoid revisiting this issue, EBC option ROM size is only limited by available system memory. EFI memory allocation services allow device drivers to claim as much memory as they need, within limits of available system memory.

The PCI specification limits the size of an image stored in an option ROM to 16 MB. If the driver is stored on the hard drive then the 16MB option ROM limit does not apply. In addition, the PE/COFF object format limits the size of images to 2 GB.
22.2 Memory Ordering

The term memory ordering refers to the order in which a processor issues reads (loads) and writes (stores) out onto the bus to system memory. The EBC Virtual Machine enforces strong memory ordering, where reads and writes are issued on the system bus in the order they occur in the instruction stream under all circumstances.

22.3 Virtual Machine Registers

The EBC virtual machine utilizes a simple register set. There are two categories of VM registers: general purpose registers and dedicated registers. All registers are 64-bits wide. There are eight (8) general-purpose registers (R0-R7), which are used by most EBC instructions to manipulate or fetch data. Table 22-1 lists the general-purpose registers in the VM and the conventions for their usage during execution.

Table 22-1 General Purpose VM Registers

<table>
<thead>
<tr>
<th>Index</th>
<th>Register</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>R0</td>
<td>Points to the top of the stack</td>
</tr>
<tr>
<td>1-3</td>
<td>R1-R3</td>
<td>Preserved across calls</td>
</tr>
<tr>
<td>4-7</td>
<td>R4-R7</td>
<td>Scratch, not preserved across calls</td>
</tr>
</tbody>
</table>

Register R0 is used as a stack pointer and is used by the CALL, RET, PUSH, and POP instructions. The VM initializes this register to point to the incoming arguments when an EBC image is started or entered. This register may be modified like any other general purpose VM register using EBC instructions. Register R7 is used for function return values.

Unlike the general-purpose registers, the VM dedicated registers have specific purposes. There are two dedicated registers: the instruction pointer (IP), and the flags (Flags) register. Specialized instructions provide access to the dedicated registers. These instructions reference the particular dedicated register by its assigned index value. Table 22-2 lists the dedicated registers and their corresponding index values.

Table 22-2 Dedicated VM Registers

<table>
<thead>
<tr>
<th>Index</th>
<th>Register</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>FLAGS</td>
<td>Bit Description</td>
</tr>
<tr>
<td></td>
<td></td>
<td>0 C = Condition code</td>
</tr>
<tr>
<td></td>
<td></td>
<td>1 SS = Single step</td>
</tr>
<tr>
<td></td>
<td></td>
<td>2..63 Reserved</td>
</tr>
<tr>
<td>1</td>
<td>IP</td>
<td>Points to current instruction</td>
</tr>
<tr>
<td>2..7</td>
<td>Reserved</td>
<td>Not defined</td>
</tr>
</tbody>
</table>

The VM Flags register contains VM status and context flags. Table 22-3 lists the descriptions of the bits in the Flags register.
Table 22-3 VM Flags Register

<table>
<thead>
<tr>
<th>Bit</th>
<th>Flag</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>C</td>
<td>Condition code. Set to 1 if the result of the last compare was true, or set</td>
</tr>
<tr>
<td></td>
<td></td>
<td>to 0 if the last compare was false. Used by conditional JMP instructions.</td>
</tr>
<tr>
<td>1</td>
<td>S</td>
<td>Single-step. If set, causes the VM to generate a single-step exception</td>
</tr>
<tr>
<td></td>
<td></td>
<td>after executing each instruction. The bit is not cleared by the VM</td>
</tr>
<tr>
<td></td>
<td></td>
<td>following the exception.</td>
</tr>
<tr>
<td>2..63</td>
<td>-</td>
<td>Reserved</td>
</tr>
</tbody>
</table>

The VM IP register is used as an instruction pointer and holds the address of the currently executing EBC instruction. The virtual machine will update the IP to the address of the next instruction on completion of the current instruction, and will continue execution from the address indicated in IP. The IP register can be moved into any general-purpose register (R0-R7). Data manipulation and data movement instructions can then be used to manipulate the value. The only instructions that may modify the IP are the JMP, CALL, and RET instructions. Since the instruction set is designed to use words as the minimum instruction entity, the low order bit (bit 0) of IP is always cleared to 0. If a JMP, CALL, or RET instruction causes bit 0 of IP to be set to 1, then an alignment exception occurs.

22.4 Natural Indexing

The natural indexing mechanism is the critical functionality that enables EBC to be executed unchanged on 32- or 64-bit systems. Natural indexing is used to specify the offset of data relative to a base address. However, rather than specifying the offset as a fixed number of bytes, the offset is encoded in a form that specifies the actual offset in two parts: a constant offset, and an offset specified as a number of natural units (where one natural unit = sizeof (VOID *)). These two values are used to compute the actual offset to data at runtime. When the VM decodes an index during execution, the resultant offset is computed based on the natural processor size. The encoded indexes themselves may be 16, 32, or 64 bits in size. Table 22-4 describes the fields in a natural index encoding.

Table 22-4 Index Encoding

<table>
<thead>
<tr>
<th>Bit #</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>N</td>
<td>Sign bit (sign), most significant bit</td>
</tr>
<tr>
<td>N-3..N-1</td>
<td>Bits assigned to natural units (w)</td>
</tr>
<tr>
<td>A..N-4</td>
<td>Constant units (c)</td>
</tr>
<tr>
<td>0..A-1</td>
<td>Natural units (n)</td>
</tr>
</tbody>
</table>

As shown in Table 22-4, for a given encoded index, the most significant bit (bit N) specifies the sign of the resultant offset after it has been calculated. The sign bit is followed by three bits (N-3..N-1) that are used to compute the width of the natural units field (n). The value (w) from this field is multiplied by the index size in bytes to determine the actual width (A) of the natural units field (n). Once the width of the natural units field has been determined, then the natural units (n) and constant units (c) can be extracted. The offset is then calculated at runtime according to the following equation:

$$\text{Offset} = (c + n \times \text{sizeof (VOID *)))} \times \text{sign}$$
The following sections describe each of these fields in more detail.

22.4.1 Sign Bit
The sign bit determines the sign of the index once the offset calculation has been performed. All index computations using “n” and “c” are done with positive numbers, and the sign bit is only used to set the sign of the final offset computed.

22.4.2 Bits Assigned to Natural Units
This 3-bit field that is used to determine the width of the natural units field. The units vary based on the size of the index according to Table 22-5. For example, for a 16-bit index, the value contained in this field would be multiplied by 2 to get the actual width of the natural-units field.

Table 22-5 Index Size in Index Encoding

<table>
<thead>
<tr>
<th>Index Size</th>
<th>Units</th>
</tr>
</thead>
<tbody>
<tr>
<td>16 bits</td>
<td>2 bits</td>
</tr>
<tr>
<td>32 bits</td>
<td>4 bits</td>
</tr>
<tr>
<td>64 bits</td>
<td>8 bits</td>
</tr>
</tbody>
</table>

22.4.3 Constant
The constant is the number of bytes in the index that do not scale with processor size. When the index is a 16-bit value, the maximum constant is 4095. This index is achieved when the bits assigned to natural units is 0.

22.4.4 Natural Units
Natural units are used when a structure has fields that can vary with the architecture of the processor. Fields that precipitate the use of natural units include pointers and EFI INTN and UINTN data types. The size of one pointer or INTN/UINTN equals one natural unit. The natural units field in an index encoding is a count of the number of natural fields whose sizes (in bytes) must be added to determine a field offset.

As an example, assume that a given EBC instruction specifies a 16-bit index of 0xA048. This breaks down into:

- Sign bit (bit 15) = 1 (negative offset)
- Bits assigned to natural units (w, bits 14-12) = 2. Multiply by index size in bytes = 2 x 2 = 4 (A)
- c = bits 11-4 = 4
- n = bits 3-0 = 8

On a 32-bit machine, the offset is then calculated to be:
- Offset = (4 + 8 * 4) * -1 = -36
- On a 64-bit machine, the offset is calculated to be:
- Offset = (4 + 8 * 8) * -1 = -68
22.5 EBC Instruction Operands

The VM supports an EBC instruction set that performs data movement, data manipulation, branching, and other miscellaneous operations typical of a simple processor. Most instructions operate on two operands, and have the general form:

```
INSTRUCTION Operand1, Operand2
```

Typically, instruction operands will be one of the following:

- Direct
- Indirect
- Indirect with index
- Immediate

The following subsections explain these operands.

22.5.1 Direct Operands

When a direct operand is specified for an instruction, the data to operate upon is contained in one of the VM general-purpose registers \( R0-R7 \). Syntactically, an example of direct operand mode could be the \( \text{ADD} \) instruction:

```
ADD64 R1, R2
```

This form of the instruction utilizes two direct operands. For this particular instruction, the VM would take the contents of register \( R2 \), add it to the contents of register \( R1 \), and store the result in register \( R1 \).

22.5.2 Indirect Operands

When an indirect operand is specified, a VM register contains the address of the operand data. This is sometimes referred to as register indirect, and is indicated by prefixing the register operand with “@.” Syntactically, an example of an indirect operand mode could be this form of the \( \text{ADD} \) instruction:

```
ADD32 R1, @R2
```

For this instruction, the VM would take the 32-bit value at the address specified in \( R2 \), add it to the contents of register \( R1 \), and store the result in register \( R1 \).

22.5.3 Indirect with Index Operands

When an indirect with index operand is specified, the address of the operand is computed by adding the contents of a register to a decoded natural index that is included in the instruction. Typically with indexed addressing, the base address will be loaded in the register and an index value will be used to indicate the offset relative to this base address. Indexed addressing takes the form

```
@R1 (+n, +c)
```

where:

- \( R1 \) is one of the general-purpose registers (\( R0-R7 \)) which contains the base address
- \( +n \) is a count of the number of “natural” units offset. This portion of the total offset is computed at runtime as \( n * \text{sizeof(VOID*)} \)
• \( +c \) is a byte offset to add to the natural offset to resolve the total offset.

The values of \( n \) and \( c \) can be either positive or negative, though they must both have the same sign. These values get encoded in the indexes associated with EBC instructions as shown in Table 22-4. Indexes can be 16-, 32-, or 64-bits wide depending on the instruction. An example of indirect with index syntax would be:

\[
\text{ADD32 R1, @R2 (+1, +8)}
\]

This instruction would take the address in register \( R2 \), add \((8 + 1 \times \text{sizeof (VOID*)})\), read the 32-bit value at the address, add the contents of \( R1 \) to the value, and store the result back to \( R1 \).

### 22.5.4 Immediate Operands

Some instructions support an immediate operand, which is simply a value included in the instruction encoding. The immediate value may or may not be sign extended, depending on the particular instruction. One instruction that supports an immediate operand is \texttt{MOVI}. An example usage of this instruction is:

\[
\text{MOVIww R1, 0x1234}
\]

This instruction moves the immediate value 0x1234 directly into VM register \( R1 \). The immediate value is contained directly in the encoding for the MOVI instruction.

### 22.6 EBC Instruction Syntax

Most EBC instructions have one or more variations that modify the size of the instruction and/or the behavior of the instruction itself. These variations will typically modify an instruction in one or more of the following ways:

- The size of the data being operated upon
- The addressing mode for the operands
- The size of index or immediate data
- To represent these variations syntactically in this specification the following conventions are used:
  - Natural indexes are indicated with the “Index” keyword, and may take the form of “Index16,” “Index32,” or “Index64” to indicate the size of the index value supported. Sometimes the form Index16|32|64 is used here, which is simply a shorthand notation for Index16|Index32|Index64. A natural index is encoded per Table 22-4 and is resolved at runtime.
  - Immediate values are indicated with the “Immed” keyword, and may take the form of “Immed16,” “Immed32,” or “Immed64” to indicate the size of the immediate value supported. The shorthand notation Immed16|32|64 is sometimes used when different size immediate values are supported.
  - Terms in brackets \([ \) ] are required.
  - Terms in braces \{ \} are optional.
  - Alternate terms are separated by a vertical bar \|\).
• The form R₁ and R₂ represent Operand 1 register and Operand 2 register respectfully, and can typically be any VM general-purpose register R₀-R₇.
• Within descriptions of the instructions, brackets [ ] enclosing a register and/or index indicate that the contents of the memory pointed to by the enclosed contents are used.

22.7 Instruction Encoding

Most EBC instructions take the form:

\[ \text{INSTRUCTION } R₁, \ R₂ \ \text{Index|Immed} \]

For those instructions that adhere to this form, the binary encoding for the instruction will typically consist of an opcode byte, followed by an operands byte, followed by two or more bytes of immediate or index data. Thus the instruction stream will be:

\((1 \text{ Byte Opcode}) + (1 \text{ Byte Operands}) + (\text{Immediate data|Index data})\)

22.7.1 Instruction Opcode Byte Encoding

The first byte of an instruction is the opcode byte, and an instruction’s actual opcode value consumes 6 bits of this byte. The remaining two bits will typically be used to indicate operand sizes and/or presence or absence of index or immediate data. Table 22-6 defines the bits in the opcode byte for most instructions, and their usage.

<table>
<thead>
<tr>
<th>Bit</th>
<th>Sym</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>6..7</td>
<td>Modifiers</td>
<td>One or more of: \Index or immediate data present/absent\ Operand size Index or immediate data size</td>
</tr>
<tr>
<td>0..5</td>
<td>Op</td>
<td>Instruction opcode</td>
</tr>
</tbody>
</table>

For those instructions that use bit 7 to indicate the presence of an index or immediate data and bit 6 to indicate the size of the index or immediate data, if bit 7 is 0 (no immediate data), then bit 6 is ignored by the VM. Otherwise, unless otherwise specified for a given instruction, setting unused bits in the opcode byte results in an instruction encoding exception when the instruction is executed. Setting the modifiers field in the opcode byte to reserved values will also result in an instruction encoding exception.

22.7.2 Instruction Operands Byte Encoding

The second byte of most encoded instructions is an operand byte, which encodes the registers for the instruction operands and whether the operands are direct or indirect. Table 22-7 defines the encoding for the operand byte for these instructions. Unless otherwise specified for a given instruction, setting unused bits in the operand byte results in an instruction encoding exception when the instruction is executed. Setting fields in the operand byte to reserved values will also result in an instruction encoding exception.
22.7.3 Index/Immediate Data Encoding
Following the operand bytes for most instructions is the instruction’s immediate data. The immediate data is, depending on the instruction and instruction encoding, either an unsigned or signed literal value, or an index encoded using natural encoding. In either case, the size of the immediate data is specified in the instruction encoding.

For most instructions, the index/immediate value in the instruction stream is interpreted as a signed immediate value if the register operand is direct. This immediate value is then added to the contents of the register to compute the instruction operand. If the register is indirect, then the data is usually interpreted as a natural index (see Section 22.4) and the computed index value is added to the contents of the register to get the address of the operand.

22.8 EBC Instruction Set
The following sections describe each of the EBC instructions in detail. Information includes an assembly-language syntax, a description of the instruction functionality, binary encoding, and any limitations or unique behaviors of the instruction.

ADD

Syntax

\[
\text{ADD}[32|64] \ {\@}R_1, \ {\@}R_2 \ \{\text{Index16}|\text{Immed16}\}
\]

Description

Adds two signed operands and stores the result to Operand 1. The operation can be performed on either 32-bit (ADD32) or 64-bit (ADD64) operands.
Operation

**Operand 1 <= Operand 1 + Operand 2**

Table 22-8 ADD Instruction Encoding

<table>
<thead>
<tr>
<th>BYTE</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>Bit Description</td>
</tr>
</tbody>
</table>
| 7    | 0 = Immediate/index absent  
|      | 1 = Immediate/index present |
| 6    | 0 = 32-bit operation  
|      | 1 = 64-bit operation |
| 0..5 | Opcode = 0x0C |
| 1    | Bit Description |
| 7    | 0 = Operand 2 direct  
|      | 1 = Operand 2 indirect |
| 4..6 | Operand 2 |
| 3    | 0 = Operand 1 direct  
|      | 1 = Operand 1 indirect |
| 0..2 | Operand 1 |
| 2..3 | Optional 16-bit immediate data/index |

Behaviors and Restrictions

- If Operand 2 is indirect, then the immediate data is interpreted as an index and the Operand 2 value is fetched from memory as a signed value at address \([R_2 + \text{Index16}]\).
- If Operand 2 is direct, then the immediate data is considered a signed immediate value and is added to the \(R_2\) register contents such that \(\text{Operand 2} = \text{R}_2 + \text{Immed16}\).
- If the instruction is ADD32 and Operand 1 is direct, then the result is stored back to the Operand 1 register with the upper 32 bits cleared.

AND

**Syntax**

\[
\text{AND}[32|64] \{@\text{R}_1\}, \{@\text{R}_2\} \{\text{Index16}|\text{Immed16}\}
\]

**Description**

Performs a logical AND operation on two operands and stores the result to Operand 1. The operation can be performed on either 32-bit (AND32) or 64-bit (AND64) operands.
Operation

**Operand 1 <= Operand 1 AND Operand 2**

Table 22-9 AND Instruction Encoding

<table>
<thead>
<tr>
<th>BYTE</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>Bit Description</td>
</tr>
<tr>
<td>7</td>
<td>0 = Immediate/index absent</td>
</tr>
<tr>
<td></td>
<td>1 = Immediate/index present</td>
</tr>
<tr>
<td>6</td>
<td>0 = 32-bit operation</td>
</tr>
<tr>
<td></td>
<td>1 = 64-bit operation</td>
</tr>
<tr>
<td>0..5</td>
<td>Opcode = 0x14</td>
</tr>
<tr>
<td>1</td>
<td>Bit Description</td>
</tr>
<tr>
<td>7</td>
<td>0 = Operand 2 direct</td>
</tr>
<tr>
<td></td>
<td>1 = Operand 2 indirect</td>
</tr>
<tr>
<td>4..6</td>
<td>Operand 2</td>
</tr>
<tr>
<td>3</td>
<td>0 = Operand 1 direct</td>
</tr>
<tr>
<td></td>
<td>1 = Operand 1 indirect</td>
</tr>
<tr>
<td>0..2</td>
<td>Operand 1</td>
</tr>
<tr>
<td>2..3</td>
<td>Optional 16-bit immediate data/index</td>
</tr>
</tbody>
</table>

Behaviors and Restrictions

- If Operand 2 is indirect, then the immediate data is interpreted as an index, and the Operand 2 value is fetched from memory as an unsigned value at address [R2 + Index16].
- If Operand 2 is direct, then the immediate data is considered a signed immediate value and is added to the register contents such that Operand 2 = R2 + Immed16.
- If the instruction is AND32 and Operand 1 is direct, then the result is stored to the Operand 1 register with the upper 32 bits cleared.

**ASHR**

Syntax

`ASHR[32|64] {@}R1, {@}R2 {Index16|Immed16}`

Description

Performs an arithmetic right-shift of a signed 32-bit (ASHR32) or 64-bit (ASHR64) operand and stores the result back to Operand 1.
Operation

Operand 1 <= Operand 1 SHIFT-RIGHT Operand 2

Table 22-10 ASHR Instruction Encoding

<table>
<thead>
<tr>
<th>BYTE</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>Bit</td>
</tr>
<tr>
<td></td>
<td>0 = Immediate/index absent</td>
</tr>
<tr>
<td></td>
<td>1 = Immediate/index present</td>
</tr>
<tr>
<td>7</td>
<td>0 = 32-bit operation</td>
</tr>
<tr>
<td></td>
<td>1 = 64-bit operation</td>
</tr>
<tr>
<td>6</td>
<td>Opcode = 0x19</td>
</tr>
<tr>
<td>0..5</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>Bit</td>
</tr>
<tr>
<td></td>
<td>0 = Operand 2 direct</td>
</tr>
<tr>
<td></td>
<td>1 = Operand 2 indirect</td>
</tr>
<tr>
<td>7</td>
<td>0 = Operand 1 direct</td>
</tr>
<tr>
<td></td>
<td>1 = Operand 1 indirect</td>
</tr>
<tr>
<td>4..6</td>
<td>Operand 2</td>
</tr>
<tr>
<td>3</td>
<td>0 = Operand 1 direct</td>
</tr>
<tr>
<td></td>
<td>1 = Operand 1 indirect</td>
</tr>
<tr>
<td>0..2</td>
<td>Operand 1</td>
</tr>
<tr>
<td>2..3</td>
<td>Optional 16-bit immediate data/index</td>
</tr>
</tbody>
</table>

Behaviors and Restrictions

- If Operand 2 is indirect, then the immediate data is interpreted as an index, and the Operand 2 value is fetched from memory as a signed value at address \([R_2 + \text{Index16}]\).
- If Operand 2 is direct, then the immediate data is considered a signed immediate value and is added to the register contents such that \(\text{Operand 2} = R_2 + \text{Immed16}\).
- If the instruction is ASHR32, and Operand 1 is direct, then the result is stored back to the Operand 1 register with the upper 32 bits cleared.

BREAK

Syntax

BREAK [break code]

Description

The BREAK instruction is used to perform special processing by the VM. The break code specifies the functionality to perform.

BREAK 0 – Runaway program break. This indicates that the VM is likely executing code from cleared memory. This results in a bad break exception.

BREAK 1 – Get virtual machine version. This instruction returns the 64-bit virtual machine revision number in VM register R7. The encoding is shown in Table 22-11 and Table 22-12. A VM that conforms to this version of the specification should return a version number of 0x00010000.
Table 22-11 VM Version Format

<table>
<thead>
<tr>
<th>Bits</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>63-32</td>
<td>Reserved = 0</td>
</tr>
<tr>
<td>31..16</td>
<td>VM major version</td>
</tr>
<tr>
<td>15..0</td>
<td>VM minor version</td>
</tr>
</tbody>
</table>

**BREAK 3** – Debug breakpoint. Executing this instruction results in a debug break exception. If a debugger is attached or available, then it may halt execution of the image.

**BREAK 4** – System call. There are no system calls supported for use with this break code, so the VM will ignore the instruction and continue execution at the following instruction.

**BREAK 5** – Create thunk. This causes the interpreter to create a thunk for the EBC entry point whose 32-bit IP-relative offset is stored at the 64-bit address in VM register R7. The interpreter then replaces the contents of the memory location pointed to by R7 to point to the newly created thunk. Since all EBC IP-relative offsets are relative to the next instruction or data object, the original offset is off by 4, so must be incremented by 4 to get the actual address of the entry point.

**BREAK 6** – Set compiler version. An EBC C compiler can insert this break instruction into an executable to set the compiler version used to build an EBC image. When the VM executes this instruction it takes the compiler version from register R7 and may perform version compatibility checking. The compiler version number follows the same format as the VM version number returned by the BREAK 1 instruction.

Table 22-12 BREAK Instruction Encoding

<table>
<thead>
<tr>
<th>Byte</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>Opcode = 0x00</td>
</tr>
<tr>
<td>1</td>
<td>0 = Runaway program break</td>
</tr>
<tr>
<td></td>
<td>1 = Get virtual machine version</td>
</tr>
<tr>
<td></td>
<td>3 = Debug breakpoint</td>
</tr>
<tr>
<td></td>
<td>4 = System call</td>
</tr>
<tr>
<td></td>
<td>5 = Create thunk</td>
</tr>
<tr>
<td></td>
<td>6 = Set compiler version</td>
</tr>
</tbody>
</table>

**Behaviors and Restrictions**

- Executing an undefined BREAK code results in a bad break exception.
- Executing BREAK 0 results in a bad break exception.
CALL

Syntax

CALL32{EX}{a} {@}R1 {Immed32|Index32}
CALL64{EX}{a} Immed64

Description

The CALL instruction pushes the address of the following instruction on the stack and jumps to a subroutine. The subroutine may be either EBC or native code, and may be to an absolute or IP-relative address. CALL32 is used to jump directly to EBC code within a given application, whereas CALLEX is used to jump to external code (either native or EBC), which requires thunking. Functionally, the CALL does the following:

```
R0 = R0 - 8;
PUSH64 ReturnAddress
if (Opcode.ImmedData64Bit) {
  if (Operands.EbcCall) {
    IP = Immed64;
  } else {
    NativeCall (Immed64);
  }
} else {
  if (Operand1 != R0) {
    Addr = Operand1;
  } else {
    Addr = Immed32;
  }
  if (Operands.EbcCall) {
    if (Operands.RelativeAddress) {
      IP += Addr + SizeOfThisInstruction;
    } else {
      IP = Addr
    }
  } else {
    if (Operands.RelativeAddress) {
      NativeCall (IP + Addr)
    } else {
      NativeCall (Addr)
    }
  }
}
```

Operation

```
R0 <= R0 - 16
[R0] <= IP + SizeOfThisInstruction
IP <= IP + SizeOfThisInstruction + Operand 1 (relative CALL)
IP <= Operand 1 (absolute CALL)
```
BEHAVIOR AND RESTRICTIONS

- For the CALL32 forms, if Operand 1 is indirect, then the immediate data is interpreted as an index, and the Operand 1 value is fetched from memory address \([R_1 + \text{Index32}]\).

- For the CALL32 forms, if Operand 1 is direct, then the immediate data is considered a signed immediate value and is added to the Operand 1 register contents such that Operand 1 = \(R_1 + \text{Immed32}\).

- For the CALLEX forms, the VM must fix up the stack pointer and execute a call to native code in a manner compatible with the native code such that the callee is able to access arguments passed on the VM stack.

- For the CALLEX forms, the value returned by the callee should be returned in \(R7\).

- For the CALL64 forms, the Operand 1 fields are ignored.

- If Byte7:Bit6 = 1 (CALL64), then Byte1:Bit4 is assumed to be 0 (absolute address)

- For CALL32 forms, if Operand 1 register = \(R0\), then the register operand is ignored and only the immediate data is used in the calculation of the call address.

- Prior to the call, the VM will decrement the stack pointer \(R0\) by 16 bytes, and store the 64-bit return address on the stack.

- Offsets for relative calls are relative to the address of the instruction following the CALL instruction.

---

**Table 22-13 CALL Instruction Encoding**

<table>
<thead>
<tr>
<th>BYTE</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>Bit Description</td>
</tr>
</tbody>
</table>
| 7    | 0 = Immediate/index data absent  
1 = Immediate/index data present |
| 6    | 0 = CALL32 with 32-bit immediate data/index if present  
1 = CALL64 with 64-bit immediate data |
| 0..5 | Opcode = 0x03 |
| 1    | Bit Description |
| 6..7 | Reserved = 0 |
| 5    | 0 = Call to EBC  
1 = Call to native code |
| 4    | 0 = Absolute address  
1 = Relative address |
| 3    | 0 = Operand 1 direct  
1 = Operand 1 indirect |
| 0..2 | Operand 1 |
| 2..5 | Optional 32-bit index/immediate for CALL32 |
| 2..9 | Required 64-bit immediate data for CALL64 |
CMP

Syntax

```
CMP[32|64][eq|lte|gte|ulte|ugte] R1, {@}R2 {Index16|Immed16}
```

Description

The CMP instruction is used to compare Operand 1 to Operand 2. Supported comparison modes are =, <=, >=, unsigned <=, and unsigned >=. The comparison size can be 32 bits (CMP32) or 64 bits (CMP64). The effect of this instruction is to set or clear the condition code bit in the Flags register per the comparison results. The operands are compared as signed values except for the CMPulte and CMPugte forms.

Operation

- **CMPeq**: Flags.C <= (Operand 1 == Operand 2)
- **CMPlte**: Flags.C <= (Operand 1 <= Operand 2)
- **CMPgte**: Flags.C <= (Operand 1 >= Operand 2)
- **CMPulte**: Flags.C <= (Operand 1 <= Operand 2) (unsigned)
- **CMPugte**: Flags.C <= (Operand 1>= Operand 2) (unsigned)

Table 22-14 CMP Instruction Encoding

<table>
<thead>
<tr>
<th>BYTE</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>Bit</td>
</tr>
</tbody>
</table>
| 7    | 0 = Immediate/index data absent  
1 = Immediate/index data present |
| 6    | 0 = 32-bit comparison  
1 = 64-bit comparison |
| 0..5 | Opcode      |
|      | 0x05 = CMPeq compare equal  
0x06 = CMPlte compare signed less then/equal  
0x07 = CMPgte compare signed greater than/equal  
0x08 = CMPulte compare unsigned less than/equal  
0x09 = CMPugte compare unsigned greater than/equal |
| 1    | Bit         |
| 7    | 0 = Operand 2 direct  
1 = Operand 2 indirect |
| 4..6 | Operand 2 |
| 3    | Reserved = 0 |
| 0..2 | Operand 1 |
| 2..3 | Optional 16-bit immediate data/index |
Behaviors and Restrictions

- If Operand 2 is indirect, then the immediate data is interpreted as an index, and the Operand 2 value is fetched from memory address [R2 + Index16].
- If Operand 2 is direct, then the immediate data is considered a signed immediate value and is added to the register contents such that Operand 2 = R2 + Immed16.
- Only register direct is supported for Operand 1.

CMPI

Syntax

CMPI{32|64}{w|d}{eq|lte|gte|ulte|ugte} {@}R1 {Index16}, Immed16|Immed32

Description

Compares two operands, one of which is an immediate value, for =, <=, >=, unsigned <=, or unsigned >=, and sets or clears the condition flag bit in the Flags register accordingly. Comparisons can be performed on a 32-bit (CMPI32) or 64-bit (CMPI64) basis. The size of the immediate data can be either 16 bits (CMPIw) or 32 bits (CMPId).

Operation

| CMPIeq: Flags.C <= (Operand 1 == Operand 2) |
| CMPIlte: Flags.C <= (Operand 1 <= Operand 2) |
| CMPIgte: Flags.C <= (Operand 1 >= Operand 2) |
| CMPIulte: Flags.C <= (Operand 1 <= Operand 2) |
| CMPIugte: Flags.C <= (Operand 1>= Operand 2) |

Table 22-15 CMPI Instruction Encoding

<table>
<thead>
<tr>
<th>BYTE</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>Bit</td>
</tr>
</tbody>
</table>
| 7    | 0 = 16-bit immediate data  
1 = 32-bit immediate data |
| 6    | 0 = 32-bit comparison  
1 = 64-bit comparison |
| 0..5 | Opcode      |
|      | 0x2D = CMPIeq compare equal  
0x2E = CMPIlte compare signed less then/equal  
0x2F = CMPIgte compare signed greater than/equal  
0x30 = CMPIulte compare unsigned less than/equal  
0x31 = CMPIugte compare unsigned greater than/equal |
Behaviors and Restrictions

- The immediate data is fetched as a signed value.
- If the immediate data is smaller than the comparison size, then the immediate data is sign-extended appropriately.
- If Operand 1 is direct, and an Operand 1 index is specified, then an instruction encoding exception is generated.

DIV

Syntax

```
DIV[32|64] {@}R1, {@}R2 {Index16|Immed16}
```

Description

Performs a divide operation on two signed operands and stores the result to Operand 1. The operation can be performed on either 32-bit (DIV32) or 64-bit (DIV64) operands.

Operation

```
Operand 1 <= Operand 1 / Operand 2
```

Table 22-16 DIV Instruction Encoding

<table>
<thead>
<tr>
<th>BYTE</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>Bit</td>
</tr>
</tbody>
</table>
| 7    | 0 = Immediate/index absent
|      | 1 = Immediate/index present |
| 6    | 0 = 32-bit operation
|      | 1 = 64-bit operation |
| 0..5 | Opcode = 0x10 |
Behaviors and Restrictions
- If Operand 2 is indirect, then the immediate data is interpreted as an index, and the Operand 2 value is fetched from memory as a signed value at address [R2 + Index16].
- If Operand 2 is direct, then the immediate data is considered a signed value and is added to the register contents such that Operand 2 = R2 + Immed16
- If the instruction is DIV32 form, and Operand 1 is direct, then the upper 32 bits of the result are set to 0 before storing to the Operand 1 register.
- A divide-by-0 exception occurs if Operand 2 = 0.

DIVU

Syntax
DIVU[32|64] {@}R1, {@}R2 {Index16|Immed16}

Description
Performs a divide operation on two unsigned operands and stores the result to Operand 1. The operation can be performed on either 32-bit (DIVU32) or 64-bit (DIVU64) operands.

Operation
Operand 1 \( \leftarrow \) Operand 1 \( \div \) Operand 2

Table 22-17 DIVU Instruction Encoding

<table>
<thead>
<tr>
<th>BYTE</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>Bit</td>
</tr>
<tr>
<td>7</td>
<td>0 = Immediate/index absent&lt;br&gt;1 = Immediate/index present</td>
</tr>
<tr>
<td>6</td>
<td>0 = 32-bit operation&lt;br&gt;1 = 64-bit operation</td>
</tr>
<tr>
<td>0..5</td>
<td>Opcode = 0x11</td>
</tr>
</tbody>
</table>
Behaviors and Restrictions

- If Operand 2 is indirect, then the immediate data is interpreted as an index, and the value is fetched from memory as an unsigned value at address [R2 + Index16].
- If Operand 2 is direct, then the immediate data is considered an unsigned value and is added to the Operand 2 register contents such that Operand 2 = R2 + Immed16.
- For the DIVU32 form, if Operand 1 is direct then the upper 32 bits of the result are set to 0 before storing back to the Operand 1 register.
- A divide-by-0 exception occurs if Operand 2 = 0.

**EXTNDB**

**Syntax**

```
EXTNDB[32|64] {@}R1, {@}R2 {Index16|Immed16}
```

**Description**

Sign-extend a byte value and store the result to Operand 1. The byte can be signed extended to 32 bits (EXTNDB32) or 64 bits (EXTNDB64).

**Operation**

```
Operand 1 <= (sign extended) Operand 2
```

**Table 22-18 EXTNDB Instruction Encoding**

<table>
<thead>
<tr>
<th>BYTE</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>Bit</td>
</tr>
<tr>
<td>7</td>
<td>0 = Immediate/index absent</td>
</tr>
<tr>
<td></td>
<td>1 = Immediate/index present</td>
</tr>
<tr>
<td>6</td>
<td>0 = 32-bit operation</td>
</tr>
<tr>
<td></td>
<td>1 = 64-bit operation</td>
</tr>
<tr>
<td>0..5</td>
<td>Opcode = 0x1A</td>
</tr>
</tbody>
</table>
Behaviors and Restrictions

- If Operand 2 is indirect, then the immediate data is interpreted as an index, and the byte Operand 2 value is fetched from memory as a signed value at address \([R_2 + \text{Index16}]\).
- If Operand 2 is direct, then the immediate data is considered a signed immediate value, is added to the signed-extended byte from the Operand 2 register, and the byte result is sign extended to 32 or 64 bits.
- If the instruction is EXTNDB32 and Operand 1 is direct, then the 32-bit result is stored in the Operand 1 register with the upper 32 bits cleared.

**EXTNDD**

**Syntax**

`EXTNDD[32|64] \{@\}R_1, \{@\}R_2 \{\text{Index16}|\text{Immed16}\}`

**Description**

Sign-extend a 32-bit Operand 2 value and store the result to Operand 1. The Operand 2 value can be extended to 32 bits (EXTNDD32) or 64 bits (EXTNDD64).

**Operation**

`Operand 1 \leftarrow \text{(sign extended) Operand 2}`

**Table 22-19 EXTNDD Instruction Encoding**

<table>
<thead>
<tr>
<th>BYTE</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>Bit Description</td>
</tr>
</tbody>
</table>
| 7    | 0 = Immediate/index absent  
       | 1 = Immediate/index present |
| 6    | 0 = 32-bit operation  
       | 1 = 64-bit operation |
| 0..5 | Opcode = 0x1C |
Behaviors and Restrictions

- If Operand 2 is indirect, then the immediate data is interpreted as an index, and the 32-bit value is fetched from memory as a signed value at address \([R2 + \text{Index16}]\).
- If Operand 2 is direct, then the immediate data is considered a signed immediate value such that Operand 2 = \(R2 + \text{Immed16}\), and the value is sign extended to 32 or 64 bits accordingly.
- If the instruction is EXTND32 and Operand 1 is direct, then the result is stored in the Operand 1 register with the upper 32 bits cleared.

**EXTNDW**

Syntax

\[
\text{EXTNDW}[32|64] \{@\}R1, \{@\}R2 \{\text{Index16}|\text{Immed16}\}
\]

Description

Sign-extend a 16-bit Operand 2 value and store the result back to Operand 1. The value can be signed extended to 32 bits (EXTNDW32) or 64 bits (EXTNDW64).

Operation

\[\text{Operand 1} \Leftarrow (\text{sign extended}) \text{Operand 2}\]

Table 22-20 EXTNDW Instruction Encoding

<table>
<thead>
<tr>
<th>BYTE</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>Bit Description</td>
</tr>
</tbody>
</table>
| 7    | 0 = Immediate/index absent  
1 = Immediate/index present  |
| 6    | 0 = 32-bit operation   
1 = 64-bit operation     |
| 0..5 | Opcode = 0x1B         |
Behaviors and Restrictions

- If Operand 2 is indirect, then the immediate data is interpreted as an index, and the word value is fetched from memory as a signed value at address \([R_2 + \text{Index16}]\).
- If Operand 2 is direct, then the immediate data is considered a signed immediate value such that Operand 2 = \(R_2 + \text{Immed16}\), and the value is sign extended to 32 or 64 bits accordingly.
- If the instruction is EXTNDW32 and Operand 1 is direct, then the 32-bit result is stored in the Operand 1 register with the upper 32 bits cleared.

### JMP

**Syntax**

\[
\begin{align*}
\text{JMP32}\{\text{cs}\mid \text{cc}\} \ { @} R_1 \ { \text{Immed32}\mid \text{Index32}} \\
\text{JMP64}\{\text{cs}\mid \text{cc}\} \ \text{Immed64}
\end{align*}
\]

**Description**

The JMP instruction is used to conditionally or unconditionally jump to a relative or absolute address and continue executing EBC instructions. The condition test is done using the condition bit in the VM Flags register. The JMP64 form only supports an immediate value that can be used for either a relative or absolute jump. The JMP32 form adds support for indirect addressing of the JMP offset or address. The JMP is implemented as:

```c
if (ConditionMet) {
    if (Operand.RelativeJump) {
        IP += Operand1 + SizeOfThisInstruction;
    } else {
        IP = Operand1;
    }
} else {
    IP = Operand1;
}
```

**Operation**

\[
\begin{align*}
\text{IP} & \leq \text{Operand 1 (absolute address)} \\
\text{IP} & \leq \text{IP + SizeOfThisInstruction + Operand 1 (relative address)}
\end{align*}
\]
Table 22-21 JMP Instruction Encoding

<table>
<thead>
<tr>
<th>Byte</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>Bit  Description</td>
</tr>
</tbody>
</table>
| 7    | 0 = Immediate/index data absent  
      | 1 = Immediate/index data present  |
| 6    | 0 = JMP32  
      | 1 = JMP64  |
| 0..5 | Opcode = 0x01             |
| 1    | Bit  Description          |
| 7    | 0 = Unconditional jump  
      | 1 = Conditional jump      |
| 6    | 0 = Jump if Flags.C is clear (cc)  
      | 1 = Jump if Flags.C is set (cs)  |
| 5    | Reserved = 0              |
| 4    | 0 = Absolute address      
      | 1 = Relative address      |
| 3    | 0 = Operand 1 direct      
      | 1 = Operand 1 indirect    |
| 0..2 | Operand 1                  |
| 2..5 | Optional 32-bit immediate data/index for JMP32 |
| 2..9 | 64-bit immediate data for JMP64 |

Behaviors and Restrictions

- Operand 1 fields are ignored for the JMP64 forms
- If the instruction is JMP32, and Operand 1 register = R0, then the register contents are assumed to be 0.
- If the instruction is JMP32, and Operand 1 is indirect, then the immediate data is interpreted as an index, and the jump offset or address is fetched as a 32-bit signed value from address [R1 + Index32]
- If the instruction is JMP32, and Operand 1 is direct, then the immediate data is considered a signed immediate value such that Operand 1 = R1 + Immed32
- If the jump is unconditional, then Byte1:Bit6 (condition) is ignored
- If the instruction is JMP64, and Byte0:Bit7 is clear (no immediate data), then an instruction encoding exception is generated.
- If the instruction is JMP32, and Operand 2 is indirect, then the Operand 2 value is read as a natural value from memory address [R1 + Index32]
- An alignment check exception is generated if the jump is taken and the target address is odd.
JMP8

Syntax

\[ \text{JMP8}\{\text{cs|cc}\} \text{ Immed8} \]

Description

Conditionally or unconditionally jump to a relative offset and continue execution. The offset is a signed one-byte offset specified in the number of words. The offset is relative to the start of the following instruction.

Operation

\[ \text{IP} = \text{IP} + \text{SizeOfThisInstruction} + (\text{Immed8} \times 2) \]

Table 22-22 JMP8 Instruction Encoding

<table>
<thead>
<tr>
<th>BYTE</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>Bit</td>
</tr>
<tr>
<td>7</td>
<td>0 = Unconditional jump</td>
</tr>
<tr>
<td></td>
<td>1 = Conditional jump</td>
</tr>
<tr>
<td>6</td>
<td>0 = Jump if Flags.C is clear (cc)</td>
</tr>
<tr>
<td></td>
<td>1 = Jump if Flags.C is set (cs)</td>
</tr>
<tr>
<td>0..5</td>
<td>Opcode = 0x02</td>
</tr>
<tr>
<td>1</td>
<td>Immediate data (signed word offset)</td>
</tr>
</tbody>
</table>

Behaviors and Restrictions

- If the jump is unconditional, then Byte0:Bit6 (condition) is ignored

LOADSP

Syntax

\[ \text{LOADSP} \text{ [Flags], R2} \]

Description

This instruction loads a VM dedicated register with the contents of a VM general-purpose register R0-R7. The dedicated register is specified by its index as shown in Table 22-2.
Operation

\[
\text{Operand 1} \leftarrow R_2
\]

Table 22-23 LOADSP Instruction Encoding

<table>
<thead>
<tr>
<th>BYTE</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>Bit Description</td>
</tr>
<tr>
<td>6..7</td>
<td>Reserved = 0</td>
</tr>
<tr>
<td>0..5</td>
<td>Opcode = 0x29</td>
</tr>
<tr>
<td>1</td>
<td>7 Reserved</td>
</tr>
<tr>
<td>4..6</td>
<td>Operand 2 general purpose register</td>
</tr>
<tr>
<td>3</td>
<td>Reserved</td>
</tr>
<tr>
<td>0..2</td>
<td>Operand 1 dedicated register index</td>
</tr>
</tbody>
</table>

Behaviors and Restrictions

- Attempting to load any register (Operand 1) other than the Flags register results in an instruction encoding exception.
- Specifying a reserved dedicated register index results in an instruction encoding exception.
- If Operand 1 is the Flags register, then reserved bits in the Flags register are not modified by this instruction.

MOD

Syntax

\[
\text{MOD}[32|64] \{@\!R_1, \{@\!R_2 \{\text{Index16|Immed16}\}
\]

Description

Perform a modulus on two signed 32-bit (MOD32) or 64-bit (MOD64) operands and store the result to Operand 1.

Operation

\[
\text{Operand 1} \leftarrow \text{Operand 1} \text{ MOD } \text{Operand 2}
\]

Table 22-24 MOD Instruction Encoding

<table>
<thead>
<tr>
<th>BYTE</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>Bit Description</td>
</tr>
<tr>
<td>7</td>
<td>0 = Immediate/index absent</td>
</tr>
<tr>
<td></td>
<td>1 = Immediate/index present</td>
</tr>
<tr>
<td>6</td>
<td>0 = 32-bit operation</td>
</tr>
<tr>
<td></td>
<td>1 = 64-bit operation</td>
</tr>
<tr>
<td>0..5</td>
<td>Opcode = 0x12</td>
</tr>
</tbody>
</table>
Behaviors and Restrictions

- If Operand 2 is indirect, then the immediate data is interpreted as an index, and the Operand 2 value is fetched from memory as a signed value at address \([R_2 + \text{Index16}]\).
- If Operand 2 is direct, then the immediate data is considered a signed immediate value such that Operand 2 = \(R_2 + \text{Immed16}\), and the value is sign extended to 32 or 64 bits accordingly.
- If Operand 2 = 0, then a divide-by-zero exception is generated.

MODU

Syntax

\[
\text{MODU}[32|64] \{@\}R_1, \{@\}R_2 \{\text{Index16}|\text{Immed16}\}
\]

Description

Perform a modulus on two unsigned 32-bit (MODU32) or 64-bit (MODU64) operands and store the result to Operand 1.

Operation

\[
\text{Operand 1} \leftarrow \text{Operand 1} \mod \text{Operand 2}
\]

Table 22-25 MODU Instruction Encoding

<table>
<thead>
<tr>
<th>BYTE</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>Bit Description</td>
</tr>
</tbody>
</table>
| 7    | 0 = Immediate/index absent  
      | 1 = Immediate/index present |
| 6    | 0 = 32-bit operation  
      | 1 = 64-bit operation |
| 0..5 | Opcode = 0x13 |
Behaviors and Restrictions

- If Operand 2 is indirect, then the immediate data is interpreted as an index, and the Operand 2 value is fetched from memory as an unsigned value at address \([R2 + \text{Index16}]\).
- If Operand 2 is direct, then the immediate data is considered an unsigned immediate value such that \(\text{Operand 2} = R2 + \text{Immed16}\).
- If \(\text{Operand 2} = 0\), then a divide-by-zero exception is generated.

**MOV**

**Syntax**

\[
\begin{align*}
\text{MOV}[b|w|d|q]\{w|d\} \{@\}R1 \{\text{Index16|32}\}, \{@\}R2 \{\text{Index16|32}\} \\
\text{MOVqq} \{@\}R1 \{\text{Index64}\}, \{@\}R2 \{\text{Index64}\}
\end{align*}
\]

**Description**

This instruction moves data from Operand 2 to Operand 1. Both operands can be indexed, though both indexes are the same size. In the instruction syntax for the first form, the first variable character indicates the size of the data move, which can be 8 bits (b), 16 bits (w), 32 bits (d), or 64 bits (q). The optional character indicates the presence and size of the index value(s), which may be 16 bits (w) or 32 bits (d). The MOVqq instruction adds support for 64-bit indexes.

**Operation**

\(\text{Operand 1} \leftarrow \text{Operand 2}\)

**Table 22-26 MOV Instruction Encoding**

<table>
<thead>
<tr>
<th>Byte</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Bit</td>
</tr>
<tr>
<td>0</td>
<td>0 = Operand 2 direct</td>
</tr>
<tr>
<td></td>
<td>1 = Operand 2 indirect</td>
</tr>
<tr>
<td>7</td>
<td>0 = Operand 1 direct</td>
</tr>
<tr>
<td></td>
<td>1 = Operand 1 indirect</td>
</tr>
<tr>
<td>6</td>
<td>0 = Operand 2 index absent</td>
</tr>
<tr>
<td></td>
<td>1 = Operand 2 index present</td>
</tr>
</tbody>
</table>
Behaviors and Restrictions

- If an index is specified for Operand 1, and Operand 1 is direct, then an instruction encoding exception is generated.

**MOVI**

**Syntax**

\[ \text{MOVI[b|w|d|q][w|d|q]} \{@\}R1 \{Index16\}, \text{Immed16|32|64} \]

**Description**

This instruction moves a signed immediate value to Operand 1. In the instruction syntax, the first variable character specifies the width of the move, which may be 8 bits (b), 16 bits (w), 32-bits (d), or 64 bits (q). The second variable character specifies the width of the immediate data, which may be 16 bits (w), 32 bits (d), or 64 bits (q).

**Operation**

\[ \text{Operand 1} \leftarrow \text{Operand 2} \]
Table 22-27 MOVI Instruction Encoding

<table>
<thead>
<tr>
<th>BYTE</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>Bit Description</td>
</tr>
</tbody>
</table>
| 6..7 | 0 = Reserved  
1 = Immediate data is 16 bits (w)  
2 = Immediate data is 32 bits (d)  
3 = Immediate data is 64 bits (q) |
| 0..5 | Opcode = 0x37 |
| 1    | Bit Description |
| 7    | Reserved = 0 |
| 6    | 0 = Operand 1 index absent  
1 = Operand 1 index present |
| 4..5 | 0 = 8 bit (b) move  
1 = 16 bit (w) move  
2 = 32 bit (d) move  
3 = 64 bit (q) move |
| 3    | 0 = Operand 1 direct  
1 = Operand 1 indirect |
| 0..2 | Operand 1 |
| 2..3 | Optional 16-bit index |
| 2..3/4..5 | 16-bit immediate data |
| 2..5/4..7 | 32-bit immediate data |
| 2..9/4..11 | 64-bit immediate data |

Behaviors and Restrictions
• Specifying an index value with Operand 1 direct results in an instruction encoding exception.
• If the immediate data is smaller than the move size, then the value is sign-extended to the width of the move.
• If Operand 1 is a register, then the value is stored to the register with bits beyond the move size cleared.

MOVI

Syntax

\[ \text{MOVI}[w|d|q] \ \{@\}R1 \ \{Index16\}, \ Index16|32|64 \]

Description
This instruction moves an indexed value of form (+n, +c) to Operand 1. The index value is converted from (+n, +c) format to a signed offset per the encoding described in Table 22-4. The size of the Operand 2 index data can be 16 (w), 32 (d), or 64 (q) bits.
Operation

**Operand 1 <= Operand 2 (index value)**

Table 22-28 MOVIn Instruction Encoding

<table>
<thead>
<tr>
<th>BYTE</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>Bit Description</td>
</tr>
</tbody>
</table>
| 6..7 | 0 = Reserved  
1 = Operand 2 index value is 16 bits (w)  
2 = Operand 2 index value is 32 bits (d)  
3 = Operand 2 index value is 64 bits (q) |
| 0..5 | Opcode = 0x38 |
| 1    | Bit Description |
| 7    | Reserved |
| 6    | 0 = Operand 1 index absent  
1 = Operand 1 index present |
| 4..5 | Reserved = 0 |
| 3    | 0 = Operand 1 direct  
1 = Operand 1 indirect |
| 0..2 | Operand 1 |
| 2..3 | Optional 16-bit Operand 1 index |
| 2..3/4..5 | 16-bit Operand 2 index |
| 2..5/4..7 | 32-bit Operand 2 index |
| 2..9/4..11 | 64-bit Operand 2 index |

Behaviors and Restrictions

- Specifying an Operand 1 index when Operand 1 is direct results in an instruction encoding exception.
- The Operand 2 index is sign extended to the size of the move if necessary.
- If the Operand 2 index size is smaller than the move size, then the value is truncated.
- If Operand 1 is direct, then theOperand 2 value is sign extended to 64 bits and stored to the Operand 1 register.

**MOVn**

**Syntax**

\[
MOVn{w|d} \{@\}R1 \{Index16|32\}, \{@\}R2 \{Index16|32\}
\]

**Description**

This instruction loads an unsigned natural value from Operand 2 and stores the value to Operand 1. Both operands can be indexed, though both operand indexes are the same size. The operand index(s) can be 16 bits (w) or 32 bits (d).
Operation

\[
\text{Operand1} \leftarrow (\text{UINTN})\text{Operand2}
\]

Table 22-29 MOVn Instruction Encoding

<table>
<thead>
<tr>
<th>BYTE</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>Bit</td>
</tr>
</tbody>
</table>
| 7    | 0 = Operand 1 index absent  
      | 1 = Operand 1 index present |
| 6    | 0 = Operand 2 index absent  
      | 1 = Operand 2 index present |
| 0..5 | 0x32 = MOVnw opcode  
      | 0x33 = MOVnd opcode |
| 1    | Bit         |
| 7    | 0 = Operand 2 direct  
      | 1 = Operand 2 indirect |
| 4..6 | Operand 2    |
| 3    | 0 = Operand 1 direct  
      | 1 = Operand 1 indirect |
| 0..2 | Operand 1    |
| 2..3 | Optional Operand 1 16-bit index |
| 2..3/4..5 | Optional Operand 2 16-bit index |
| 2..5 | Optional Operand 1 32-bit index |
| 2..5/6..9 | Optional Operand 2 32-bit index |

Behaviors and Restrictions

- If an index is specified for Operand 2, and Operand 2 register is direct, then the Operand 2 index value is added to the register contents such that Operand 2 = (UINTN)(R2 + Index).
- If an index is specified for Operand 1, and Operand 1 is direct, then an instruction encoding exception is generated.
- If Operand 1 is direct, then the Operand 2 value will be 0-extended to 64 bits on a 32-bit machine before storing to the Operand 1 register.

MOVREL

Syntax

\[
\text{MOVREL}[w|d|q] \{@\}R1 \{\text{Index16}\}, \text{Immed16}\{32|64\}
\]

Description

This instruction fetches data at an IP-relative immediate offset (Operand 2) and stores the result to Operand 1. The offset is a signed offset relative to the following instruction. The fetched data is unsigned and may be 16 (w), 32 (d), or 64 (q) bits in size.
Operation

\[
\text{Operand 1} \leftarrow [\text{IP} + \text{SizeOfThisInstruction} + \text{Immed}]
\]

Table 22-30 MOVREL Instruction Encoding

<table>
<thead>
<tr>
<th>BYTE</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>Bit Description</td>
</tr>
</tbody>
</table>
| 6..7 | 0 = Reserved  
     | 1 = Immediate data is 16 bits (w)  
     | 2 = Immediate data is 32 bits (d)  
     | 3 = Immediate data is 64 bits (q)  |
| 0..5 | Opcode = 0x39 |
| 1    | Bit Description |
| 7    | Reserved = 0 |
| 6    | 0 = Operand 1 index absent  
     | 1 = Operand 1 index present |
| 4..5 | Reserved = 0 |
| 3    | 0 = Operand 1 direct  
     | 1 = Operand 1 indirect |
| 0..2 | Operand 1 |
| 2..3 | Optional 16-bit Operand 1 index |
| 2..3/4..5 | 16-bit immediate offset |
| 2..5/4..7 | 32-bit immediate offset |
| 2..9/4..11 | 64-bit immediate offset |

Behaviors and Restrictions

- If an Operand 1 index is specified and Operand 1 is direct, then an instruction encoding exception is generated.

MOVsn

Syntax

\[
\text{MOVsn}\{w\} \{R_1\}, \{Index16\}, \{R_2\} \{Index16|Immed16\} \\
\text{MOVsn}\{d\} \{R_1\} \{Index32\}, \{R_2\} \{Index32|Immed32\}
\]

Description

Moves a signed natural value from Operand 2 to Operand 1. Both operands can be indexed, though the indexes are the same size. Indexes can be either 16 bits (MOVsnw) or 32 bits (MOVsnd) in size.
Operation

\[ \text{Operand 1} \leq \text{Operand 2} \]

Table 22-31 MOVsn Instruction Encoding

<table>
<thead>
<tr>
<th>BYTE</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>Bit</td>
</tr>
</tbody>
</table>
| 7    | 0 = Operand 1 index absent  
9 = Operand 1 index present |
| 6    | 0 = Operand 2 index/immediate data absent  
1 = Operand 2 index/immediate data present |
| 0..5 | 0x25 = MOVsnw opcode  
0x26 = MOVsnd opcode |
| 1    | Bit         |
| 7    | 0 = Operand 2 direct  
1 = Operand 2 indirect |
| 4..6 | Operand 2 |
| 3    | 0 = Operand 1 direct  
1 = Operand 1 indirect |
| 0..2 | Operand 1  |
| 2..3 | Optional 16-bit Operand 1 index (MOVsnw)  |
| 2..3/4..5 | Optional 16-bit Operand 2 index (MOVsnw)  |
| 2..5 | Optional 32-bit Operand 1 index/immediate data (MOVsnd)  |
| 2..5/6..9 | Optional 32-bit Operand 2 index/immediate data (MOVsnd)  |

Behaviors and Restrictions

- If Operand 2 is direct, and Operand 2 index/immediate data is specified, then the immediate value is read as a signed immediate value and is added to the contents of Operand 2 register such that Operand 2 = R2 + Immed.
- If Operand 2 is indirect, and Operand 2 index/immediate data is specified, then the immediate data is interpreted as an index and the Operand 2 value is fetched from memory as a signed value at address [R2 + Index16].
- If an index is specified for Operand 1, and Operand 1 is direct, then an instruction encoding exception is generated.
- If Operand 1 is direct, then the Operand 2 value is sign-extended to 64-bits on 32-bit native machines.
MUL

Syntax

\texttt{MUL[32|64] \{@\}R1, \{@\}R2 \{Index16|Immed16\}}

Description
Perform a signed multiply of two operands and store the result back to Operand 1. The operands can be either 32 bits (MUL32) or 64 bits (MUL64).

Operation

\texttt{Operand 1 \Leftarrow Operand \ast Operand 2}

Table 22-32 MUL Instruction Encoding

<table>
<thead>
<tr>
<th>BYTE</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>Bit Description</td>
</tr>
</tbody>
</table>
| 7    | 0 = Operand 2 immediate/index absent  
               1 = Operand 2 immediate/index present |
| 6    | 0 = 32-bit operation  
               1 = 64-bit operation |
| 0..5 | Opcode = 0x0E |
| 1    | Bit Description |
| 7    | 0 = Operand 2 direct  
               1 = Operand 2 indirect |
| 4..6 | Operand 2 |
| 3    | 0 = Operand 1 direct  
               1 = Operand 1 indirect |
| 0..2 | Operand 1 |
| 2..3 | Optional 16-bit Operand 2 immediate data/index |

Behaviors and Restrictions

- If Operand 2 is indirect, then the immediate data is interpreted as an index, and the Operand 2 value is fetched from memory as a signed value at address \([R_2 + \text{Index16}].

- If Operand 2 is direct, then the immediate data is considered a signed immediate value and is added to the Operand 2 register contents such that Operand 2 = R_2 + \text{Immed16}.

- If the instruction is MUL32, and Operand 1 is direct, then the result is stored to Operand 1 register with the upper 32 bits cleared.
MULU

Syntax

MULU[32|64] {@}R1, {@}R2 {Index16|Immed16}

Description

Performs an unsigned multiply of two 32-bit (MULU32) or 64-bit (MULU64) operands, and stores the result back to Operand 1.

Operation

Operand 1 <= Operand * Operand 2

Table 22-33 MULU Instruction Encoding

<table>
<thead>
<tr>
<th>BYTE</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>Bit Description</td>
</tr>
<tr>
<td>7</td>
<td>0 = Operand 2 immediate/index absent</td>
</tr>
<tr>
<td></td>
<td>1 = Operand 2 immediate/index present</td>
</tr>
<tr>
<td>6</td>
<td>0 = 32-bit operation</td>
</tr>
<tr>
<td></td>
<td>1 = 64-bit operation</td>
</tr>
<tr>
<td></td>
<td>0..5 Opcode = 0x0F</td>
</tr>
<tr>
<td>1</td>
<td>Bit Description</td>
</tr>
<tr>
<td>7</td>
<td>0 = Operand 2 direct</td>
</tr>
<tr>
<td></td>
<td>1 = Operand 2 indirect</td>
</tr>
<tr>
<td>4..6</td>
<td>Operand 2</td>
</tr>
<tr>
<td>3</td>
<td>0 = Operand 1 direct</td>
</tr>
<tr>
<td></td>
<td>1 = Operand 1 indirect</td>
</tr>
<tr>
<td>0..2</td>
<td>Operand 1</td>
</tr>
<tr>
<td>2..3</td>
<td>Optional 16-bit immediate data/index</td>
</tr>
</tbody>
</table>

Behaviors and Restrictions

- If Operand 2 is indirect, then the immediate data is interpreted as an index, and the Operand 2 value is fetched from memory as an unsigned value at address [R2 + Index16].
- If Operand 2 is direct, then the immediate data is considered a signed immediate value and is added to the Operand 2 register contents such that Operand 2 = R2 + Immed16.
- If the instruction is MULU32 and Operand 1 is direct, then the result is written to the Operand 1 register with the upper 32 bits cleared.
NEG

Syntax

```
NEG[32|64] {@}R1, {@}R2 {Index16|Immed16}
```

Description

Multiply Operand 2 by negative 1, and store the result back to Operand 1.Operand 2 is a signed value and fetched as either a 32-bit (NEG32) or 64-bit (NEG64) value.

Operation

```
Operand 1 <= -1 * Operand 2
```

Table 22-34 NEG Instruction Encoding

<table>
<thead>
<tr>
<th>BYTE</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>Bit</td>
</tr>
<tr>
<td>7</td>
<td>0 = Operand 2 immediate/index absent</td>
</tr>
<tr>
<td></td>
<td>1 = Operand 2 immediate/index present</td>
</tr>
<tr>
<td>6</td>
<td>0 = 32-bit operation</td>
</tr>
<tr>
<td></td>
<td>1 = 64-bit operation</td>
</tr>
<tr>
<td>0..5</td>
<td>Opcode = 0x0B</td>
</tr>
<tr>
<td>1</td>
<td>Bit</td>
</tr>
<tr>
<td>7</td>
<td>0 = Operand 2 direct</td>
</tr>
<tr>
<td></td>
<td>1 = Operand 2 indirect</td>
</tr>
<tr>
<td>4..6</td>
<td>Operand 2</td>
</tr>
<tr>
<td>3</td>
<td>0 = Operand 1 direct</td>
</tr>
<tr>
<td></td>
<td>1 = Operand 1 indirect</td>
</tr>
<tr>
<td>0..2</td>
<td>Operand 1</td>
</tr>
<tr>
<td>2..3</td>
<td>Optional 16-bit immediate data/index</td>
</tr>
</tbody>
</table>

Behaviors and Restrictions

- If Operand 2 is indirect, then the immediate data is interpreted as an index, and the Operand 2 value is fetched from memory as a signed value at address [R2 + Index16].
- If Operand 2 is direct, then the immediate data is considered a signed immediate value and is added to the Operand 2 register contents such that Operand 2 = R2 + Immed16.
- If the instruction is NEG32 and Operand 1 is direct, then the result is stored in Operand 1 register with the upper 32-bits cleared.
NOT

Syntax
NOT[32|64] {@}R1, {@}R2 {Index16|Immed16}

Description
Performs a logical NOT operation on Operand 2, an unsigned 32-bit (NOT32) or 64-bit (NOT64) value, and stores the result back to Operand 1.

Operation
Operand 1 <= NOT Operand 2

Table 22-35 NOT Instruction Encoding

<table>
<thead>
<tr>
<th>BYTE</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>Bit Description</td>
</tr>
<tr>
<td>7</td>
<td>0 = Operand 2 immediate/index absent</td>
</tr>
<tr>
<td></td>
<td>1 = Operand 2 immediate/index present</td>
</tr>
<tr>
<td>6</td>
<td>0 = 32-bit operation</td>
</tr>
<tr>
<td></td>
<td>1 = 64-bit operation</td>
</tr>
<tr>
<td>0..5</td>
<td>Opcode = 0x0A</td>
</tr>
<tr>
<td>1</td>
<td>Bit Description</td>
</tr>
<tr>
<td>7</td>
<td>0 = Operand 2 direct</td>
</tr>
<tr>
<td></td>
<td>1 = Operand 2 indirect</td>
</tr>
<tr>
<td>4..6</td>
<td>Operand 2</td>
</tr>
<tr>
<td>3</td>
<td>0 = Operand 1 direct</td>
</tr>
<tr>
<td></td>
<td>1 = Operand 1 indirect</td>
</tr>
<tr>
<td>0..2</td>
<td>Operand 1</td>
</tr>
<tr>
<td>2..3</td>
<td>Optional 16-bit immediate data/index</td>
</tr>
</tbody>
</table>

Behaviors and Restrictions
- If Operand 2 is indirect, then the immediate data is interpreted as an index, and the Operand 2 value is fetched from memory as an unsigned value at address \([R_2 + Index16]\).
- If Operand 2 is direct, then the immediate data is considered a signed immediate value and is added to the Operand 2 register contents such that Operand 2 = R_2 + Immed16.
- If the instruction is NOT32 and Operand 1 is a register, then the result is stored in the Operand 1 register with the upper 32 bits cleared.
OR

Syntax

$$\text{OR}^{[32|64]} \{@\}R_1, \{@\}R_2 \{\text{Index16|Immed16}\}$$

Description
Performs a bit-wise OR of two 32-bit (OR32) or 64-bit (OR64) operands, and stores the result back to Operand 1.

Operation

$$\text{Operand 1} \leftarrow \text{Operand 1 OR Operand 2}$$

Table 22-36 OR Instruction Encoding

<table>
<thead>
<tr>
<th>BYTE</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>Bit Description</td>
</tr>
</tbody>
</table>
| 7    | 0 = Operand 2 immediate/index absent  
        1 = Operand 2 immediate/index present |
| 6    | 0 = 32-bit operation  
        1 = 64-bit operation |
| 0..5 | Opcode = 0x15 |
| 1    | Bit Description |
| 7    | 0 = Operand 2 direct  
        1 = Operand 2 indirect |
| 4..6 | Operand 2 |
| 3    | 0 = Operand 1 direct  
        1 = Operand 1 indirect |
| 0..2 | Operand 1 |
| 2..3 | Optional 16-bit immediate data/index |

Behaviors and Restrictions

- If Operand 2 is indirect, then the immediate data is interpreted as an index, and the Operand 2 value is fetched from memory as an unsigned value at address $$[R_2 + \text{Index16}]$$.

- If Operand 2 is direct, then the immediate data is considered a signed immediate value and is added to the Operand 2 register contents such that $$\text{Operand 2} = R_2 + \text{Immed16}$$.

- If the instruction is OR32 and Operand 1 is direct, then the result is stored to Operand 1 register with the upper 32 bits cleared.
POP

Syntax

\[ \text{POP[32|64]} \{\text{@}\}R_1 \{\text{Index16}\|\text{Immed16}\} \]

Description

This instruction pops a 32-bit (POP32) or 64-bit (POP64) value from the stack, stores the result to Operand 1, and adjusts the stack pointer \( R_0 \) accordingly.

Operation

\[
\begin{align*}
\text{Operand 1} & \leftarrow [R_0] \\
R_0 & \leftarrow R_0 + 4 \quad \text{(POP32)} \\
R_0 & \leftarrow R_0 + 8 \quad \text{(POP64)}
\end{align*}
\]

Table 22-37 POP Instruction Encoding

<table>
<thead>
<tr>
<th>BYTE</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>Bit Description</td>
</tr>
</tbody>
</table>
| 7    | 0 = Immediate/index absent  
|      | 1 = Immediate/index present |
| 6    | 0 = 32-bit operation  
|      | 1 = 64-bit operation |
| 0..5 | Opcode = 0x2C |
| 1    | Bit Description |
| 7..4 | Reserved = 0 |
| 3    | 0 = Operand 1 direct  
|      | 1 = Operand 1 indirect |
| 0..2 | Operand 1 |
| 2..3 | Optional 16-bit immediate data/index |

Behaviors and Restrictions

- If Operand 1 is direct, and an index/immediate data is specified, then the immediate data is read as a signed value and is added to the value popped from the stack, and the result stored to the Operand 1 register.
- If Operand 1 is indirect, then the immediate data is interpreted as an index, and the value popped from the stack is stored to address \([R_1 + \text{Index16}]\).
- If the instruction is POP32, and Operand 1 is direct, then the popped value is sign-extended to 64 bits before storing to the Operand 1 register.
POPn

Syntax

\[ \text{POPn } \{\text{@}R1 \{\text{Index16|Immed16}\} \]

Description

Read an unsigned natural value from memory pointed to by stack pointer \( R0 \), adjust the stack pointer accordingly, and store the value back to Operand 1.

Operation

\[
\begin{align*}
\text{Operand 1} & \Leftarrow (\text{UINTN})[R0] \\
R0 & \Leftarrow R0 + \text{sizeof (VOID *)}
\end{align*}
\]

Table 22-38 POPn Instruction Encoding

<table>
<thead>
<tr>
<th>BYTE</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>Bit Description</td>
</tr>
<tr>
<td>7</td>
<td>0 = Immediate/index absent</td>
</tr>
<tr>
<td></td>
<td>1 = Immediate/index present</td>
</tr>
<tr>
<td>6</td>
<td>Reserved = 0</td>
</tr>
<tr>
<td>0..5</td>
<td>Opcode = 0x36</td>
</tr>
<tr>
<td>1</td>
<td>Bit Description</td>
</tr>
<tr>
<td>7..4</td>
<td>Reserved = 0</td>
</tr>
<tr>
<td>3</td>
<td>0 = Operand 1 direct</td>
</tr>
<tr>
<td></td>
<td>1 = Operand 1 indirect</td>
</tr>
<tr>
<td>0..2</td>
<td>Operand 1</td>
</tr>
<tr>
<td>2..3</td>
<td>Optional 16-bit immediate data/index</td>
</tr>
</tbody>
</table>

Behaviors and Restrictions

- If Operand 1 is direct, and an index/immediate data is specified, then the immediate data is fetched as a signed value and is added to the value popped from the stack and the result is stored back to the Operand 1 register.
- If Operand 1 is indirect, and an index/immediate data is specified, then the immediate data is interpreted as a natural index and the value popped from the stack is stored at \([R1 + \text{Index16}]\).
- If Operand 1 is direct, and the instruction is executed on a 32-bit machine, then the result is stored to the Operand 1 register with the upper 32 bits cleared.
PUSH

Syntax

\[ \text{PUSH}^{[32|64]} \{@\text{R1} \{\text{Index16}|\text{Immed16}\} \]

Description
Adjust the stack pointer \( \text{R0} \) and store a 32-bit (PUSH32) or 64-bit (PUSH64) Operand 1 value on the stack.

Operation

\[ \begin{align*}
\text{R0} & \leftarrow \text{R0} - 4 \quad (\text{PUSH32}) \\
\text{R0} & \leftarrow \text{R0} - 8 \quad (\text{PUSH64}) \\
[\text{R0}] & \leftarrow \text{Operand 1}
\end{align*} \]

Table 22-39 PUSH Instruction Encoding

<table>
<thead>
<tr>
<th>BYTE</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>Bit</td>
</tr>
</tbody>
</table>
| 7    | 0 = Immediate/index absent  
       | 1 = Immediate/index present |
| 6    | 0 = 32-bit operation  
       | 1 = 64-bit operation |
| 0.5  | Opcode = 0x2B |
| 1    | Bit         |
| 7..4 | Reserved = 0 |
| 3    | 0 = Operand 1 direct  
       | 1 = Operand 1 indirect |
| 0..2 | Operand 1 |
| 2..3 | Optional 16-bit immediate data/index |

Behaviors and Restrictions

- If Operand 1 is direct, and an index/immediate data is specified, then the immediate data is read as a signed value and is added to the Operand 1 register contents such that Operand 1 = \( \text{R1} + \text{Immed16} \).
- If Operand 1 is indirect, and an index/immediate data is specified, then the immediate data is interpreted as a natural index and the pushed value is read from \([\text{R1} + \text{Index16}]\).

PUSHn

Syntax

\[ \text{PUSHn} \{@\text{R1} \{\text{Index16}|\text{Immed16}\} \]

Description
Adjust the stack pointer \( \text{R0} \), and store a natural value on the stack.
Operation

\[
R0 \leftarrow R0 - \text{sizeof (VOID *)}
\]

\[
[R0] \leftarrow \text{Operand 1}
\]

Table 22-40 PUSHn Instruction Encoding

<table>
<thead>
<tr>
<th>BYTE</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>Bit Description</td>
</tr>
<tr>
<td>7</td>
<td>0 = Immediate/index absent</td>
</tr>
<tr>
<td></td>
<td>1 = Immediate/index present</td>
</tr>
<tr>
<td>6</td>
<td>Reserved = 0</td>
</tr>
<tr>
<td>0..5</td>
<td>Opcode = 0x35</td>
</tr>
<tr>
<td>1</td>
<td>Bit Description</td>
</tr>
<tr>
<td>7..4</td>
<td>Reserved = 0</td>
</tr>
<tr>
<td>3</td>
<td>0 = Operand 1 direct</td>
</tr>
<tr>
<td></td>
<td>1 = Operand 1 indirect</td>
</tr>
<tr>
<td>0..2</td>
<td>Operand 1</td>
</tr>
<tr>
<td>2..3</td>
<td>Optional 16-bit immediate data/index</td>
</tr>
</tbody>
</table>

Behaviors and Restrictions

- If Operand 1 is direct, and an index/immediate data is specified, then the immediate data is fetched as a signed value and is added to the Operand 1 register contents such that Operand 1 = R1 + Immed16.

- If Operand 1 is indirect, and an index/immediate data is specified, then the immediate data is interpreted as a natural index and the Operand 1 value pushed is fetched from [R1 + Index16].

RET

Syntax

\[
\text{RET}
\]

Description

This instruction fetches the return address from the stack, sets the IP to the value, adjusts the stack pointer register R0, and continues execution at the return address. If the RET is a final return from the EBC driver, then execution control returns to the caller, which may be EBC or native code.

Operation

\[
\text{IP} \leftarrow [R0]
\]

\[
R0 \leftarrow R0 + 16
\]
Behaviors and Restrictions
- An alignment exception will be generated if the return address is not aligned on a 16-bit boundary.

**SHL**

**Syntax**

\[
\text{SHL}[32|64] \{@R_1, \{@R_2 \{\text{Index16|Immed16}}
\]

**Description**

Left-shifts Operand 1 by Operand 2 bit positions and stores the result back to Operand 1. The operand sizes may be either 32-bits (SHL32) or 64 bits (SHL64).

**Operation**

\[\text{Operand 1} \ll \text{Operand 2} \]

**Table 22-42 SHL Instruction Encoding**

<table>
<thead>
<tr>
<th>BYTE</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>Bit</td>
</tr>
<tr>
<td>7</td>
<td>0 = Operand 2 immediate/index absent</td>
</tr>
<tr>
<td></td>
<td>1 = Operand 2 immediate/index present</td>
</tr>
<tr>
<td>6</td>
<td>0 = 32-bit operation</td>
</tr>
<tr>
<td></td>
<td>1 = 64-bit operation</td>
</tr>
<tr>
<td>0..5</td>
<td>Opcode = 0x17</td>
</tr>
<tr>
<td>1</td>
<td>Bit</td>
</tr>
<tr>
<td>7</td>
<td>0 = Operand 2 direct</td>
</tr>
<tr>
<td></td>
<td>1 = Operand 2 indirect</td>
</tr>
<tr>
<td>4..6</td>
<td>Operand 2</td>
</tr>
<tr>
<td>3</td>
<td>0 = Operand 1 direct</td>
</tr>
<tr>
<td></td>
<td>1 = Operand 1 indirect</td>
</tr>
<tr>
<td>0..2</td>
<td>Operand 1</td>
</tr>
<tr>
<td>2..3</td>
<td>Optional 16-bit immediate data/index</td>
</tr>
</tbody>
</table>
Behaviors and Restrictions

- If Operand 2 is indirect, then the immediate data is interpreted as an index, and the Operand 2 value is fetched from memory as an unsigned value at address [R2 + Index16].

- If Operand 2 is direct, then the immediate data is considered a signed immediate value and is added to the Operand 2 register contents such that Operand 2 = R2 + Immed16.

- If the instruction is SHL32, and Operand 1 is direct, then the result is stored to the Operand 1 register with the upper 32 bits cleared.

**SHR**

**Syntax**

```
SHR[32|64] {@}R1, {@}R2 {Index16|Immed16}
```

**Description**

Right-shifts unsigned Operand 1 by Operand 2 bit positions and stores the result back to Operand 1. The operand sizes may be either 32-bits (SHR32) or 64 bits (SHR64).

**Operation**

```
Operand 1 <= Operand 1 >> Operand 2
```

**Table 22-43 SHR Instruction Encoding**

<table>
<thead>
<tr>
<th>BYTE</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>Bit Description</td>
</tr>
</tbody>
</table>
| 7    | 0 = Operand 2 immediate/index absent  
|      | 1 = Operand 2 immediate/index present  |
| 6    | 0 = 32-bit operation  
|      | 1 = 64-bit operation            |
| 0..5 | Opcode = 0x18                   |
| 1    | Bit Description                 |
| 7    | 0 = Operand 2 direct  
|      | 1 = Operand 2 indirect          |
| 4..6 | Operand 2                      |
| 3    | 0 = Operand 1 direct  
|      | 1 = Operand 1 indirect          |
| 0..2 | Operand 1                      |
| 2..3 | Optional 16-bit immediate data/index |

Behaviors and Restrictions

- If Operand 2 is indirect, then the immediate data is interpreted as an index, and the Operand 2 value is fetched from memory as an unsigned value at address [R2 + Index16].

- If Operand 2 is direct, then the immediate data is considered a signed immediate value and is added to the Operand 2 register contents such that Operand 2 = R2 + Immed16.
• If the instruction is SHR32, and Operand 1 is direct, then the result is stored to the Operand 1 register with the upper 32 bits cleared.

STORESP

Syntax

STORESP R1, [IP|Flags]

Description

This instruction transfers the contents of a dedicated register to a general-purpose register. See Table 22-2 for the VM dedicated registers and their corresponding index values.

Operation

Operand 1 <= Operand 2

Table 22-44 STORESP Instruction Encoding

<table>
<thead>
<tr>
<th>BYTE</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>Bit Description</td>
</tr>
<tr>
<td></td>
<td>6..7 Reserved = 0</td>
</tr>
<tr>
<td></td>
<td>0..5 Opcode = 0x2A</td>
</tr>
<tr>
<td>1</td>
<td>7</td>
</tr>
<tr>
<td></td>
<td>4..6 Operand 2 dedicated register index</td>
</tr>
<tr>
<td></td>
<td>3</td>
</tr>
<tr>
<td></td>
<td>0..2 Operand 1 general purpose register</td>
</tr>
</tbody>
</table>

Behaviors and Restrictions

• Specifying an invalid dedicated register index results in an instruction encoding exception.

SUB

Syntax

SUB[32|64] {@}R1, {@}R2 {Index16|Immed16}

Description

Subtracts a 32-bit (SUB32) or 64-bit (SUB64) signed Operand 2 value from a signed Operand 1 value of the same size, and stores the result to Operand 1.
Operation

\[
\text{Operand 1} \leftarrow \text{Operand 1} - \text{Operand 2}
\]

Table 22-45 SUB Instruction Encoding

<table>
<thead>
<tr>
<th>BYTE</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>Bit Description</td>
</tr>
</tbody>
</table>
| 7    | 0 = Operand 2 immediate/index absent  
1 = Operand 2 immediate/index present |
| 6    | 0 = 32-bit operation        
1 = 64-bit operation          |
| 0..5 | Opcode = 0x0D                |

Behaviors and Restrictions

- If Operand 2 is indirect, then the immediate data is interpreted as an index, and the Operand 2 value is fetched from memory as a signed value at address [R2 + Index16].
- If Operand 2 is direct, then the immediate data is considered a signed immediate value and is added to the Operand 2 register contents such that Operand 2 = R2 + Immed16.
- If the instruction is SUB32 and Operand 1 is direct, then the result is stored to the Operand 1 register with the upper 32 bits cleared.

XOR

Syntax

\[
\text{XOR[32|64]} \{@\text{R}_1\}, \{@\text{R}_2\} \{\text{Index16|Immed16}\}
\]

Description

Performs a bit-wise exclusive OR of two 32-bit (XOR32) or 64-bit (XOR64) operands, and stores the result back to Operand 1.
Operation

Operand 1 <= Operand 1 XOR Operand 2

Table 22-46 XOR Instruction Encoding

<table>
<thead>
<tr>
<th>BYTE</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>Bit Description</td>
</tr>
<tr>
<td>7</td>
<td>0 = Operand 2 immediate/index absent</td>
</tr>
<tr>
<td></td>
<td>1 =Operand 2 immediate/index present</td>
</tr>
<tr>
<td>6</td>
<td>0 = 32-bit operation</td>
</tr>
<tr>
<td></td>
<td>1 = 64-bit operation</td>
</tr>
<tr>
<td>0..5</td>
<td>Opcode = 0x16</td>
</tr>
<tr>
<td>1</td>
<td>Bit Description</td>
</tr>
<tr>
<td>7</td>
<td>0 = Operand 2 direct</td>
</tr>
<tr>
<td></td>
<td>1 = Operand 2 indirect</td>
</tr>
<tr>
<td>4..6</td>
<td>Operand 2</td>
</tr>
<tr>
<td>3</td>
<td>0 = Operand 1 direct</td>
</tr>
<tr>
<td></td>
<td>1 = Operand 1 indirect</td>
</tr>
<tr>
<td>0..2</td>
<td>Operand 1</td>
</tr>
<tr>
<td>2..3</td>
<td>Optional 16-bit immediate data/index</td>
</tr>
</tbody>
</table>

Behaviors and Restrictions

- If Operand 2 is indirect, then the immediate data is interpreted as an index, and the Operand 2 value is fetched from memory as an unsigned value at address \([R_2 + \text{Index16}]\).
- If Operand 2 is direct, then the immediate data is considered a signed immediate value and is added to the Operand 2 register contents such that Operand 2 = \(R_2 + \text{Immed16}\).
- If the instruction is XOR32 and Operand1 is direct, then the result is stored to the Operand 1 register with the upper 32-bits cleared.

22.9 Runtime and Software Conventions

22.9.1 Calling Outside VM

Calls can be made to routines in other modules that are native or in another VM. It is the responsibility of the calling VM to prepare the outgoing arguments correctly to make the call outside the VM. It is also the responsibility of the VM to prepare the incoming arguments correctly for the call from outside the VM. Calls outside the VM must use the `CALL` instruction.

22.9.2 Calling Inside VM

Calls inside VM can be made either directly using the `CALL` or `CALLEX` instructions. Using direct CALL instructions is an optimization.
22.9.3 Parameter Passing
Parameters are pushed on the VM stack per the CDECL calling convention. Per this convention, the last argument in the parameter list is pushed on the stack first, and the first argument in the parameter list is pushed on the stack last.

All parameters are stored or accessed as natural size (using naturally sized instruction) except 64-bit integers, which are pushed as 64-bit values. 32-bit integers are pushed as natural size (since they should be passed as 64-bit parameter values on 64-bit machines).

22.9.4 Return Values
Return values of 8 bytes or less in size are returned in general-purpose register R7. Return values larger than 8 bytes are not supported.

22.9.5 Binary Format
PE32+ format will be used for generating binaries for the VM. A VarBss section will be included in the binary image. All global and static variables will be placed in this section. The size of the section will be based on worst-case 64-bit pointers. Initialized data and pointers will also be placed in the VarBss section, with the compiler generating code to initialize the values at runtime.

22.10 Architectural Requirements
This section provides a high level overview of the architectural requirements that are necessary to support execution of EBC on a platform.

22.10.1 EBC Image Requirements
All EBC images will be PE32+ format. Some minor additions to the format will be required to support EBC images. See the Microsoft Portable Executable and Common Object File Format Specification pointed to in Appendix Q for details of this image file format.

A given EBC image must be executable on different platforms, independent of whether it is a 32- or 64-bit processor. All EBC images should be driver implementations.

22.10.2 EBC Execution Interfacing Requirements
EBC drivers will typically be designed to execute in an (usually preboot) EFI environment. As such, EBC drivers must be able to invoke protocols and expose protocols for use by other drivers or applications. The following execution transitions must be supported:

- EBC calling EBC
- EBC calling native code
- Native code calling EBC
- Native code calling native code
- Returning from all the above transitions

Obviously native code calling native code is available by default, so is not discussed in this document.
To maintain backward compatibility with existing native code, and minimize the overhead for non-EBC drivers calling EBC protocols, all four transitions must be seamless from the application perspective. Therefore, drivers, whether EBC or native, shall not be required to have any knowledge of whether or not the calling code, or the code being called, is native or EBC compiled code. The onus is put on the tools and interpreter to support this requirement.

22.10.3 Interfacing Function Parameters Requirements
To allow code execution across protocol boundaries, the interpreter must ensure that parameters passed across execution transitions are handled in the same manner as the standard parameter passing convention for the native processor.

22.10.4 Function Return Requirements
The interpreter must support standard function returns to resume execution to the caller of external protocols. The details of this requirement are specific to the native processor. The called function must not be required to have any knowledge of whether or not the caller is EBC or native code.

22.10.5 Function Return Values Requirements
The interpreter must support standard function return values from called protocols. The exact implementation of this functionality is dependent on the native processor. This requirement applies to return values of 64 bits or less. The called function must not be required to have any knowledge of whether or not the caller is EBC or native code. Note that returning of structures is not supported.

22.11 EBC Interpreter Protocol
The EFI EBC protocol provides services to execute EBC images, which will typically be loaded into option ROMs.

EFI_EBC_PROTOCOL

Summary
This protocol provides the services that allow execution of EBC images.
GUID

#define EFI_EBC_PROTOCOL_GUID \
{0x13ac6dd1,0x73d0,0x11d4,\ 
 {0xb0,0x6b,0x00,0xaa,0x00,0xbd,0xe7}}

Protocol Interface Structure

typedef struct _EFI_EBC_PROTOCOL {
  EFI_EBC_CREATEThunk CreateThunk;
  EFI_EBC_UNLOAD_IMAGE UnloadImage;
  EFI_EBC_REGISTER_ICACHE_FLUSH RegisterICacheFlush;
  EFI_EBC_GET_VERSION GetVersion;
} EFI_EBC_PROTOCOL;

Parameters

CreateThunk  Creates a thunk for an EBC image entry point or protocol service, and returns a pointer to the thunk. See the CreateThunk() function description.

UnloadImage  Called when an EBC image is unloaded to allow the interpreter to perform any cleanup associated with the image's execution. See the UnloadImage() function description.

RegisterICacheFlush  Called to register a callback function that the EBC interpreter can call to flush the processor instruction cache after creating thunks. See the RegisterICacheFlush() function description.

GetVersion  Called to get the version of the associated EBC interpreter. See the GetVersion() function description.

Description

The EFI EBC protocol provides services to load and execute EBC images, which will typically be loaded into option ROMs. The image loader will load the EBC image, perform standard relocations, and invoke the CreateThunk() service to create a thunk for the EBC image's entry point. The image can then be run using the standard EFI start image services.

EFI_EBC_PROTOCOL.CreateThunk()

Summary

Creates a thunk for an EBC entry point, returning the address of the thunk.
Prototype

typedef
EFI_STATUS
(EIFIAPI *EFI_EBC_CREATE_THUNK) (
    IN EFI_EBC_PROTOCOL *This,
    IN EFI_HANDLE ImageHandle,
    IN VOID *EbcEntryPoint,
    OUT VOID **Thunk
);

Parameters

This A pointer to the EFI_EBC_PROTOCOL instance. This protocol is defined in Section 22.11.

ImageHandle Handle of image for which the thunk is being created.

EbcEntryPoint Address of the actual EBC entry point or protocol service the thunk should call.

Thunk Returned pointer to a thunk created.

Description

A PE32+ EBC image, like any other PE32+ image, contains an optional header that specifies the entry point for image execution. However for EBC images this is the entry point of EBC instructions, so is not directly executable by the native processor. Therefore when an EBC image is loaded, the loader must call this service to get a pointer to native code (thunk) that can be executed which will invoke the interpreter to begin execution at the original EBC entry point.

Status Codes Returned

<table>
<thead>
<tr>
<th>Code</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>EFI_SUCCESS</td>
<td>The function completed successfully.</td>
</tr>
<tr>
<td>EFI_INVALID_PARAMETER</td>
<td>Image entry point is not 2-byte aligned.</td>
</tr>
<tr>
<td>EFI_OUT_OF_RESOURCES</td>
<td>Memory could not be allocated for the thunk.</td>
</tr>
</tbody>
</table>

EFI_EBC_PROTOCOL.UnloadImage()

Summary

Called prior to unloading an EBC image from memory.
Prototype

typedef
EFI_STATUS
(EIFIAPI *EFI_EBC_UNLOAD_IMAGE) (
    IN EFI_EBC_PROTOCOL *This,
    IN EFI_HANDLE ImageHandle
);

Parameters

This A pointer to the EFI_EBC_PROTOCOL instance. This protocol is defined in Section 22.11.

ImageHandle Image handle of the EBC image that is being unloaded from memory.

Description

This function is called after an EBC image has exited, but before the image is actually unloaded. It is intended to provide the interpreter with the opportunity to perform any cleanup that may be necessary as a result of loading and executing the image.

Status Codes Returned

<table>
<thead>
<tr>
<th>Code</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>EFI_SUCCESS</td>
<td>The function completed successfully.</td>
</tr>
<tr>
<td>EFI_INVALID_PARAMETER</td>
<td>Image handle is not recognized as belonging to an EBC image that has been executed.</td>
</tr>
</tbody>
</table>

EFI_EBC_PROTOCOL.RegisterICacheFlush()

Summary

Registers a callback function that the EBC interpreter calls to flush the processor instruction cache following creation of thunks.

Prototype

typedef
EFI_STATUS
(* EFI_EBC_REGISTER_ICACHE_FLUSH) (
    IN EFI_EBC_PROTOCOL *This,
    IN EBC_ICACHE_FLUSH Flush
);

Parameters

This A pointer to the EFI_EBC_PROTOCOL instance. This protocol is defined in Section 22.11.

Flush Pointer to a function of type EBC_ICACHE_FLUSH. See “Related Definitions” below for a detailed description of this type.
Related Definitions

typedef
EFI_STATUS
(* EBC_ICACHE_FLUSH) (  
    IN EFI_PHYSICAL_ADDRESS Start,
    IN UINT64 Length
    );

Start       The beginning physical address to flush from the processor’s instruction cache.
Length      The number of bytes to flush from the processor’s instruction cache.

This is the prototype for the Flush callback routine. A pointer to a routine of this type is passed to the EBC EFI_EBC_REGISTER_ICACHE_FLUSH protocol service.

Description

An EBC image’s original PE32+ entry point is not directly executable by the native processor. Therefore to execute an EBC image, a thunk (which invokes the EBC interpreter for the image’s original entry point) must be created for the entry point, and the thunk is executed when the EBC image is started. Since the thunks may be created on-the-fly in memory, the processor’s instruction cache may require to be flushed after thunks are created. The caller to this EBC service can provide a pointer to a function to flush the instruction cache for any thunks created after the CreateThunk() service has been called. If an instruction-cache flush callback is not provided to the interpreter, then the interpreter assumes the system has no instruction cache, or that flushing the cache is not required following creation of thunks.

Status Codes Returned

<table>
<thead>
<tr>
<th>Status Code</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>EFI_SUCCESS</td>
<td>The function completed successfully.</td>
</tr>
</tbody>
</table>

EFI_EBC_PROTOCOL.GetVersion()

Summary
Called to get the version of the interpreter.

Prototype

typedef
EFI_STATUS
(* EFI_EBC_GET_VERSION) (  
    IN EFI_EBC_PROTOCOL *This,
    OUT UINT64 *Version
    );

Parameters

This                     A pointer to the EFI_EBC_PROTOCOL instance. This protocol is defined in Section 22.11.
Version                  Pointer to where to store the returned version of the interpreter.
Description
This function is called to get the version of the loaded EBC interpreter. The value and format of the returned version is identical to that returned by the EBC \texttt{BREAK 1} instruction.

Status Codes Returned

<table>
<thead>
<tr>
<th>Code</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>EFI_SUCCESS</td>
<td>The function completed successfully.</td>
</tr>
<tr>
<td>EFI_INVALID_PARAMETER</td>
<td>Version pointer is \texttt{NULL}.</td>
</tr>
</tbody>
</table>

22.12 EBC Tools

22.12.1 EBC C Compiler
This section describes the responsibilities of the EBC C compiler. To fully specify these responsibilities requires that the thunking mechanisms between EBC and native code be described.

22.12.2 C Coding Convention
The EBC C compiler supports only the C programming language. There is no support for C++, inline assembly, floating point types/operations, or C calling conventions other than CDECL.

Pointer type in C is supported only as 64-bit pointer. The code should be 64-bit pointer ready (not assign pointers to integers and vice versa).

The compiler does not support user-defined sections through pragmas.

Global variables containing pointers that are initialized will be put in the uninitialized VarBss section and the compiler will generate code to initialize these variables during load time. The code will be placed in an init text section. This compiler-generated code will be executed before the actual image entry point is executed.

22.12.3 EBC Interface Assembly Instructions
The EBC instruction set includes two forms of a \texttt{CALL} instruction that can be used to invoke external protocols. Their assembly language formats are:

\begin{verbatim}
CALLEX  Immed64
CALLEX32  @R1  {Immed32}
\end{verbatim}

Both forms can be used to invoke external protocols at an absolute address specified by the immediate data and/or register operand. The second form also supports jumping to code at a relative address. When one of these instructions is executed, the interpreter is responsible for thunking arguments and then jumping to the destination address. When the called function returns, code begins execution at the EBC instruction following the CALL instruction. The process by which this happens is called thunking. Later sections describe this operation in detail.

22.12.4 Stack Maintenance and Argument Passing
There are several EBC assembly instructions that directly manipulate the stack contents and stack pointer. These instructions operate on the EBC stack, not the interpreter stack. The instructions include the EBC \texttt{PUSH}, \texttt{POP}, \texttt{PUSHn}, and \texttt{POPN}, and all forms of the \texttt{MOV} instructions.
These instructions must adjust the EBC stack pointer in the same manner as equivalent instructions of the native instruction set. With this implementation, parameters pushed on the stack by an EBC driver can be accessed normally for stack-based native code. If native code expects parameters in registers, then the interpreter thunking process must transfer the arguments from EBC stack to the appropriate processor registers. The process would need to be reversed when native code calls EBC.

22.12.5 Native to EBC Arguments Calling Convention

The calling convention for arguments passed to EBC functions follows the standard CDECL calling convention. The arguments must be pushed as their native size. After the function arguments have been pushed on the stack, execution is passed to the called EBC function. The overhead of thunking the function parameters depends on the standard parameter passing convention for the host processor. The implementation of this functionality is left to the interpreter.

22.12.6 EBC to Native Arguments Calling Convention

When EBC makes function calls via function pointers, the EBC C compiler cannot determine whether the calls are to native code or EBC. It therefore assumes that the calls are to native code, and emits the appropriate EBC CALLEX instructions. To be compatible with calls to native code, the calling convention of EBC calling native code must follow the parameter passing convention of the native processor. The EBC C compiler generates EBC instructions that push all arguments on the stack. The interpreter is then responsible for performing the necessary thunking. The exact implementation of this functionality is left to the interpreter.

22.12.7 EBC to EBC Arguments Calling Convention

If the EBC C compiler is able to determine that a function call is to a local function, it can emit a standard EBC CALL instruction. In this case, the function arguments are passed as described in the other sections of this specification.

22.12.8 Function Returns

When EBC calls an external function, the thunking process includes setting up the host processor stack or registers such that when the called function returns, execution is passed back to the EBC at the instruction following the call. The implementation is left to the interpreter, but it must follow the standard function return process of the host processor. Typically this will require the interpreter to push the return address on the stack or move it to a processor register prior to calling the external function.

22.12.9 Function Return Values

EBC function return values of 8 bytes or less are returned in VM general-purpose register R7. Returning values larger than 8 bytes on the stack is not supported. Instead, the caller or callee must allocate memory for the return value, and the caller can pass a pointer to the callee, or the callee can return a pointer to the value in the standard return register R7.

If an EBC function returns to native code, then the interpreter thunking process is responsible for transferring the contents of R7 to an appropriate location such that the caller has access to the value using standard native code. Typically the value will be transferred to a processor register. Conversely, if a native function returns to an EBC function, the interpreter is responsible for transferring the return value from the native return memory or register location into VM register R7.
22.12.10 Thunking

Thunking is the process by which transitions between execution of native and EBC are handled. The major issues that must be addressed for thunking are the handling of function arguments, how the external function is invoked, and how return values and function returns are handled. The following sections describe the thunking process for the possible transitions.

22.12.10.1 Thunking EBC to Native Code

By definition, all external calls from within EBC are calls to native code. The EBC CALLEX instructions are used to make these calls. A typical application for EBC calling native code would be a simple “Hello World” driver. For a UEFI driver, the code could be written as shown below.

```c
EFI_STATUS EfiMain (
    IN EFI_HANDLE      ImageHandle,
    IN EFI_SYSTEM_TABLE   *ST
) {
    ST->ConOut->OutputString(ST->ConOut, L"Hello World!");
    return EFI_SUCCESS;
}
```

This C code, when compiled to EBC assembly, could result in two PUSHn instructions to push the parameters on the stack, some code to get the absolute address of the OutputString() function, then a CALLEX instruction to jump to native code. Typical pseudo assembly code for the function call could be something like the following:

```assembly
PUSHn _HelloString
PUSHn _ConOut
MOVnw R1, _OutputString
CALLEX64R1
```

The interpreter is responsible for executing the PUSHn instructions to push the arguments on the EBC stack when interpreting the PUSHn instructions. When the CALLEX instruction is encountered, it must thunk to external native code. The exact thunking mechanism is native processor dependent. For example, a supported 32-bit thunking implementation could simply move the system stack pointer to point to the EBC stack, then perform a CALL to the absolute address specified in VM register R1. However, the function calling convention for the Itanium processor family calls for the first 8 function arguments being passed in registers. Therefore, the Itanium processor family thunking mechanism requires the arguments to be copied from the EBC stack into processor registers. Then a CALL can be performed to jump to the absolute address in VM register R1. Note that since the interpreter is not aware of the number of arguments to the function being called, the maximum amount of data may be copied from the EBC stack into processor registers.

22.12.10.2 Thunking Native Code to EBC

An EBC driver may install protocols for use by other EBC drivers, or UEFI drivers or applications. These protocols provide the mechanism by which external native code can call EBC. Typical C code to install a generic protocol is shown below.
EFI_STATUS Foo(UINT32 Arg1, UINT32 Arg2);

MyProtInterface->Service1= Foo;

Status = LibInstallProtocolInterfaces (&Handle, &MyProtGUID, MyProtInterface, NULL);

To support thunking native code to EBC, the EBC compiler resolves all EBC function pointers using one level of indirection. In this way, the address of an EBC function actually becomes the address of a piece of native (thunk) code that invokes the interpreter to execute the actual EBC function. As a result of this implementation, any time the address of an EBC function is taken, the EBC C compiler must generate the following:

- A 64-bit function pointer data object that contains the actual address of the EBC function
- EBC initialization code that is executed before the image entry point that will execute EBC \texttt{BREAK} 5 instructions to create thunks for each function pointer data object
- Associated relocations for the above

So for the above code sample, the compiler must generate EBC initialization code similar to the following. This code is executed prior to execution of the actual EBC driver’s entry point.

\begin{verbatim}
MOVqq R7, Foo_pointer; get address of Foo function pointer
BREAK 5; create a thunk for the function
\end{verbatim}

The \texttt{BREAK} instruction causes the interpreter to create native thunk code elsewhere in memory, and then modify the memory location pointed to by R7 to point to the newly created thunk code for EBC function Foo. From within EBC, when the address of Foo is taken, the address of the thunk is actually returned. So for the assignment of the protocol Service1 above, the EBC C compiler will generate something like the following:

\begin{verbatim}
MOVqq R7, Foo_pointer; get address of Foo function pointer
MOVq R7, @R7; one level of indirection
MOVn R6, _MyProtInterface->Service1; get address of variable
MOVqq @R6, R7; address of thunk to ->Service1
\end{verbatim}

22.12.10.3 Thunking EBC to EBC

EBC can call EBC via function pointers or protocols. These two mechanisms are treated identically by the EBC C compiler, and are performed using EBC \texttt{CALLLEX} instructions. For EBC to call EBC, the EBC being called must have provided the address of the function. As described above, the address is actually the address of native thunk code for the actual EBC function. Therefore, when EBC calls EBC, the interpreter assumes native code is being called so prepares function arguments accordingly, and then makes the call. The native thunk code assumes native code is calling EBC, so will basically “undo” the preparation of function arguments, and then invoke the interpreter to execute the actual EBC function of interest.
22.12.11 EBC Linker

New constants must be defined for use by the linker in processing EBC images. For EBC images, the linker must set the machine type in the PE file header accordingly to indicate that the image contains EBC.

```
#define IMAGE_FILE_MACHINE_EBC 0x0EBC
```

In addition, the linker must support EBC images with of the following subsystem types as set in a PE32+ optional header:

```
#define IMAGE_SUBSYSTEM_EFI_APPLICATION       10
#define IMAGE_SUBSYSTEM_EFI_BOOT_SERVICE_DRIVER    11
#define IMAGE_SUBSYSTEM_EFI_RUNTIME_DRIVER       12
```

For EFI EBC images and object files, the following relocation types must be supported:

```
// No relocations required
#define IMAGE_REL_EBC_ABSOLUTE  0x0000
// 32-bit address w/o image base
#define IMAGE_REL_EBC_ADDR32NB  0x0001
// 32-bit relative address from byte following relocs
#define IMAGE_REL_EBC_REL32    0x0002
// Section table index
#define IMAGE_REL_EBC_SECTION   0x0003
// Offset within section
#define IMAGE_REL_EBC_SECREL   0x0004
```

The ADDR32NB relocation is used internally to the linker when RVAs are emitted. It also is used for version resources which probably will not be used. The REL32 relocation is for PC relative addressing on code. The SECTION and SECREL relocations are used for debug information.

22.12.12 Image Loader

The EFI image loader is responsible for loading an executable image into memory and applying relocation information so that an image can execute at the address in memory where it has been loaded prior to execution of the image. For EBC images, the image loader must also invoke the interpreter protocol to create a thunk for the image entry point and return the address of this thunk. After loading the image in this manner, the image can be executed in the standard manner. To implement this functionality, only minor changes will be made to EFI service EFI_BOOT_SERVICES.LoadImage(), and no changes should be made to EFI_BOOT_SERVICES.StartImage().

After the image is unloaded, the EFI image load service must call the EBC EFI_BOOT_SERVICES.UnLoadImage() service to perform any cleanup to complete unloading of the image. Typically this will include freeing up any memory allocated for thunks for the image during load and execution.

22.12.13 Debug Support

The interpreter must support debugging in an EFI environment per the EFI debug support protocol.
22.13 VM Exception Handling

This section lists the different types of exceptions that the VM may assert during execution of an EBC image. If a debugger is attached to the EBC driver via the EFI debug support protocol, then the debugger should be able to capture and identify the exception type. If a debugger is not attached, then depending on the severity of the exception, the interpreter may do one of the following:

- Invoke the EFI ASSERT() macro, which will typically display an error message and halt the system
- Sit in a while(1) loop to hang the system
- Ignore the exception and continue execution of the image (minor exceptions only)

It is a platform policy decision as to the action taken in response to EBC exceptions. The following sections describe the exceptions that may be generated by the VM.

22.13.1 Divide By 0 Exception

A divide-by-0 exception can occur for the EBC instructions DIV, DIVU, MOD, and MODU.

22.13.2 Debug Break Exception

A debug break exception occurs if the VM encounters a BREAK instruction with a break code of 3.

22.13.3 Invalid Opcode Exception

An invalid opcode exception will occur if the interpreter encounters a reserved opcode during execution.

22.13.4 Stack Fault Exception

A stack fault exception can occur if the interpreter detects that function nesting within the interpreter or system interrupts was sufficient to potentially corrupt the EBC image’s stack contents. This exception could also occur if the EBC driver attempts to adjust the stack pointer outside the range allocated to the driver.

22.13.5 Alignment Exception

An alignment exception can occur if the particular implementation of the interpreter does not support unaligned accesses to data or code. It may also occur if the stack pointer or instruction pointer becomes misaligned.

22.13.6 Instruction Encoding Exception

An instruction encoding exception can occur for the following:

- For some instructions, if an Operand 1 index is specified and Operand 1 is direct
- If an instruction encoding has reserved bits set to values other than 0
- If an instruction encoding has a field set to a reserved value.
22.13.7 Bad Break Exception

A bad break exception occurs if the VM encounters a BREAK instruction with a break code of 0, or any other unrecognized or unsupported break code.

22.13.8 Undefined Exception

An undefined exception can occur for other conditions detected by the VM. The cause of such an exception is dependent on the VM implementation, but will most likely include internal VM faults.

22.14 Option ROM Formats

The new option ROM capability is designed to be a departure from the legacy method of formatting an option ROM. PCI local bus add-in cards are the primary targets for this design although support for future bus types will be added as necessary. EFI EBC drivers can be stored in option ROMs or on hard drives in an EFI system partition.

The new format defined for the UEFI specification is intended to coexist with legacy format PCI Expansion ROM images. This provides the ability for IHVs to make a single option ROM binary that contains both legacy and new format images at the same time. This is important for the ability to have single add-in card SKUs that can work in a variety of systems both with and without native support for UEFI. Support for multiple image types in this way provides a smooth migration path during the period before widespread adoption of UEFI drivers as the primary means of support for software needed to accomplish add-in card operation in the pre-OS boot timeframe.

22.14.1 EFI Drivers for PCI Add-in Cards

The location mechanism for UEFI drivers in PCI option ROM containers is described fully in Section 11.3. Readers should refer to this section for complete details of the scheme and associated data structures.

22.14.2 Non-PCI Bus Support

EFI expansion ROMs are not supported on any other bus besides PCI local bus in the current revision of the UEFI specification.

This means that support for UEFI drivers in legacy ISA add-in card ROMs is explicitly excluded.

Support for UEFI drivers to be located on add-in card type devices for future bus designs other than PCI local bus will be added to future revisions of the UEFI specification. This support will depend upon the specifications that govern such new bus designs with respect to the mechanisms defined for support of driver code on devices.
23 - Firmware Update and Reporting

The UEFI Firmware Management Protocol provides an abstraction for device to provide firmware management support. The base requirements for managing device firmware images include identifying firmware image revision level and programming the image into the device.

The protocol for managing firmware provides the following services.

- Get the attributes of the current firmware image. Attributes include revision level.
- Get a copy of the current firmware image. As an example, this service could be used by a management application to facilitate a firmware roll-back.
- Program the device with a firmware image supplied by the user.
- Label all the firmware images within a device with a single version.

When UEFI Firmware Management Protocol (FMP) instance is intended to perform the update of an option ROM loaded from a PCI or PCI Express device, it is recommended that the FMP instance be attached to the handle with **EFI_LOADED_IMAGE_PROTOCOL** for said Option ROM.

When the FMP instance is intended to update internal device firmware, or a combination of device firmware and Option ROM, the FMP instance may instead be attached to the Controller handle of the device. However in the case where multiple devices represented by multiple controller handles are served by the same firmware store, only a single Controller handle should expose FMP. In all cases a specific updatable hardware firmware store must be represented by exactly one FMP instance.

Care should be taken to ensure that the FMP instance reports current version data that accurately represents the actual contents of the firmware store of the device exposing FMP, because in some cases the device driver currently operating the device may have been loaded from another device or media.

23.1 Firmware Management Protocol

**EFI_FIRMWARE_MANAGEMENT_PROTOCOL**

Summary

Firmware Management application invokes this protocol to manage device firmware.

GUID

```c
// {86C77A67-0B97-4633-A187-49104D0685C7}
#define EFI_FIRMWARE_MANAGEMENT_PROTOCOL_GUID \
    { 0x86c77a67, 0xb97, 0x4633, \ 
    {0xa1, 0x87, 0x49, 0x10, 0x4d, 0x06, 0x85, 0xc7 } }
```
Protocol

typedef struct _EFI_FIRMWARE_MANAGEMENT_PROTOCOL {
    EFI_FIRMWARE_MANAGEMENT_PROTOCOL_GET_IMAGE_INFO GetImageInfo;
    EFI_FIRMWARE_MANAGEMENT_PROTOCOL_GET_IMAGE GetImage;
    EFI_FIRMWARE_MANAGEMENT_PROTOCOL_SET_IMAGE SetImage;
    EFI_FIRMWARE_MANAGEMENT_PROTOCOL_CHECK_IMAGE CheckImage;
    EFI_FIRMWARE_MANAGEMENT_PROTOCOL_GET_PACKAGE_INFO GetPackageInfo;
    EFI_FIRMWARE_MANAGEMENT_PROTOCOL_SET_PACKAGE_INFO SetPackageInfo;
} EFI_FIRMWARE_MANAGEMENT_PROTOCOL;

Members

GetImageInfo
    Returns information about the current firmware image(s) of the device.
GetImage
    Retrieves a copy of the current firmware image of the device.
SetImage
    Updates the device firmware image of the device.
CheckImage
    Checks if the firmware image is valid for the device.
GetPackageInfo
    Returns information about the current firmware package.
SetPackageInfo
    Updates information about the firmware package.

EFI_FIRMWARE_MANAGEMENT_PROTOCOL.GetImageInfo()

Summary
Returns information about the current firmware image(s) of the device.
typedef EFI_STATUS
(EFIAPI *EFI_FIRMWARE_MANAGEMENT_PROTOCOL_GET_IMAGE_INFO) (  
  IN EFI_FIRMWARE_MANAGEMENT_PROTOCOL *This,  
  IN OUT UINTN *ImageInfoSize,  
  IN OUT EFI_FIRMWARE_IMAGE_DESCRIPTOR *ImageInfo,  
  OUT UINT32 *DescriptorVersion  
  OUT UINT8 *DescriptorCount,  
  OUT UINTN *DescriptorSize,  
  OUT UINT32 *PackageVersion,  
  OUT CHAR16 **PackageVersionName  
) ;

Parameters

This
A pointer to the EFI_FIRMWARE_MANAGEMENT_PROTOCOL instance.

ImageInfoSize
A pointer to the size, in bytes, of the ImageInfo buffer. On input, this is the size of the buffer allocated by the caller. On output, it is the size of the buffer returned by the firmware if the buffer was large enough, or the size of the buffer needed to contain the image(s) information if the buffer was too small.

ImageInfo
A pointer to the buffer in which firmware places the current image(s) information. The information is an array of EFI_FIRMWARE_IMAGE_DESCRIPTORs. See “Related Definitions”. May be NULL with a zero ImageInfoSize in order to determine the size of the buffer needed.

DescriptorVersion
A pointer to the location in which firmware returns the version number associated with the EFI_FIRMWARE_IMAGE_DESCRIPTOR. See “Related Definitions”.

DescriptorCount
A pointer to the location in which firmware returns the number of descriptors or firmware images within this device.

DescriptorSize
A pointer to the location in which firmware returns the size, in bytes, of an individual EFI_FIRMWARE_IMAGE_DESCRIPTOR.
PackageVersion

A version number that represents all the firmware images in the device. The format is vendor specific and new version must have a greater value than the old version. If PackageVersion is not supported, the value is 0xFFFFFFFF. A value of 0xFFFFFFFF indicates that package version comparison is to be performed using PackageVersionName. A value of 0xFFFFFFFD indicates that package version update is in progress.

PackageVersionName

A pointer to a pointer to a null-terminated string representing the package version name. The buffer is allocated by this function with AllocatePool(), and it is the caller’s responsibility to free it with a call to FreePool().

Related Definitions

```c
typedef struct {
  UINT8 ImageIndex;
  EFI_GUID ImageTypeId;
  UINT64 ImageId;
  CHAR16 *ImageIdName;
  UINT32 Version;
  CHAR16 *VersionName;
  UINTN Size;
  UINT64 AttributesSupported;
  UINT64 AttributesSetting;
  UINT64 Compatibilities;
  // Introduced with DescriptorVersion 2+
  UINT32 LowestSupportedImageVersion;
  // Introduced with DescriptorVersion 3+
  UINT32 LastAttemptVersion;
  UINT32 LastAttemptStatus;
  UINT64 HardwareInstance;
  // Introduced with DescriptorVersion 4+
  EFI_FMP_DEP *Dependencies;
} EFI_FIRMWARE_IMAGE_DESCRIPTOR;
```

ImageIndex

A unique number identifying the firmware image within the device. The number is between 1 and DescriptorCount.

ImageTypeId

A unique GUID identifying the firmware image type.

ImageId

A unique number identifying the firmware image.
**ImageIdName**
A pointer to a null-terminated string representing the firmware image name.

**Version**
Identifies the version of the device firmware. The format is vendor specific and new version must have a greater value than an old version.

**VersionName**
A pointer to a null-terminated string representing the firmware image version name.

**Size**
Size of the image in bytes. If size=0, then only ImageIndex and ImageTypeId are valid.

**AttributesSupported**
Image attributes that are supported by this device. See “Image Attribute Definitions” for possible returned values of this parameter. A value of 1 indicates the attribute is supported and the current setting value is indicated in AttributesSetting. A value of 0 indicates the attribute is not supported and the current setting value in AttributesSetting is meaningless.

**AttributesSetting**
Image attributes. See “Image Attribute Definitions” for possible returned values of this parameter.

**Compatibilities**
Image compatibilities. See “Image Compatibility Definitions” for possible returned values of this parameter.

**LowestSupportedImageVersion**
Describes the lowest ImageDescriptor version that the device will accept. Only present in version 2 or higher.

**LastAttemptVersion**
Describes the version that was last attempted to update. If no update attempted the value will be 0. If the update attempt was improperly formatted and no version number was available then the value will be zero. Only present in version 3 or higher.

**LastAttemptStatus**
Describes the status that was last attempted to update. If no update has been attempted the value will be LAST_ATTEMPT_STATUS_SUCCESS. See "Related Definitions" in Section 23.4 for Last Attempt Status values. Only present in version 3 or higher.

**HardwareInstance**
An optional number to identify the unique hardware instance within the system for devices that may have multiple instances (Example: a plug in pci network card). This number must be unique within the namespace of the ImageTypeId GUID and ImageIndex. For FMP instances that have multiple descriptors for a single hardware instance, all descriptors must have the same HardwareInstance value. This number must be consistent between boots and should be based on some sort of
hardware identified unique id (serial number, etc) whenever possible. If a hardware based number is not available the FMP provider may use some other characteristic such as device path, bus/dev/function, slot num, etc for generating the HardwareInstance. For implementations that will never have more than one instance a zero can be used. A zero means the FMP provider is not able to determine a unique hardware instance number or a hardware instance number is not needed. Only present in version 3 or higher.

Dependencies
A pointer to an array of FMP depex expression op-codes that are terminated by an EFI_FMP_DEP_END op-code.

The attribute IMAGE_ATTRIBUTE_IMAGE_UPDATABLE indicates that this device supports firmware image update.

The attribute IMAGE_ATTRIBUTE_RESET_REQUIRED indicates a reset of the device is required for the new firmware image to take effect after a firmware update. The device is the device hosting the firmware image.

The attribute IMAGE_ATTRIBUTE_AUTHENTICATION_REQUIRED indicates authentication is required to perform the following image operations: GetImage(), SetImage(), and CheckImage(). See “Image Attribute – Authentication”.

The attribute IMAGE_ATTRIBUTE_DEPENDENCY indicates that there is an EFI_FIRMWARE_IMAGE_DEP section associated with the image. See “Image Attribute – Dependency”.

Dependencies
An array of FMP depex expression op-codes that are terminated by an END op-code (see related definitions below.)
The attribute `IMAGE_ATTRIBUTE_IN_USE` indicates the current state of the firmware image. This distinguishes firmware images in a device that supports redundant images.

The attribute `IMAGE_ATTRIBUTE_UEFI_IMAGE` indicates that this image is an EFI compatible image.

```c
#define IMAGE_COMPATIBILITY_CHECK_SUPPORTED 0x0000000000000001
```

Values from 0x0000000000000002 thru 0x000000000000FFFF are reserved for future assignments.

Values from 0x0000000000010000 thru 0xFFFFFFFFFFFFFFFF are used by firmware vendor for compatibility check.

```c
#define EFI_FIRMWARE_IMAGE_DESCRIPTOR_VERSION 3
```

Description

GetImageInfo() is the only required function. GetImage(), SetImage(), CheckImage(), GetPackageInfo(), and SetPackageInfo() shall return EFI_UNSUPPORTED if not supported by the driver.
A package can have one to many firmware images. The firmware images can have the same version naming or different version naming. **PackageVersion** may be used as the representative version for all the firmware images. **PackageVersion** can be obtained from **GetPackageInfo()**. **PackageVersion** is also available in **GetImageInfo()** as **GetPackageInfo()** is optional. It also ensures the package version is in sync with the versions of the images within the package by returning the package version and image version(s) in a single function call.

The value of **ImageTypeID** is implementation specific. This feature facilitates vendor to target a single firmware release to cover multiple products within a product family. As an example, a vendor has an initial product A and then later developed a product B that is of the same product family. Product A and product B will have the same **ImageTypeID** to indicate firmware compatibility between the two products.

To determine image attributes, software must use both **AttributesSupported** and **AttributesSetting**. An attribute setting in **AttributesSetting** is meaningless if the corresponding attribute is not supported in **AttributesSupported**.

**Compatibilities** are used to ensure the targeted firmware image supports the current hardware configuration. **Compatibilities** are set based on the current hardware configuration and firmware update policy should match the current settings to those supported by the new firmware image, and only permits update to proceed if the new firmware image settings are equal or greater than the current hardware configuration settings. For example, if this function returns **Compatibilities** = 0x0000000000070001 and the new firmware image supports settings=0x0000000000030001, then the update policy should block the firmware update and notify the user that updating the hardware with the new firmware image may render the hardware inoperable. This situation usually occurs when updating the hardware with an older version of firmware.

The authentication support leverages the authentication scheme employed in variable authentication. Please reference **EFI_VARIABLE_AUTHENTICATION** in the “Variable Services” section of “Services – Runtime Services” chapter.

If **IMAGE_ATTRIBUTE_AUTHENTICATION_REQUIRED** is supported and clear, then authentication is not required to perform the firmware image operations. In firmware image operations, the image pointer points to the start of the firmware image and the image size is the firmware image.

If **IMAGE_ATTRIBUTE_AUTHENTICATION_REQUIRED** is supported and set, then authentication is required to perform the firmware image operations. In firmware image operations, the image pointer points to the start of the **authentication data** and the image size is the size of the **authentication data** and the size of the firmware image.
If `IMAGE_ATTRIBUTE_DEPENDENCY` is supported and set, then there are dependencies associated with the image. See the [Dependency Expression Instruction Set](#) for details on the format of the dependency op-codes and how they are to be used.

![Figure 23-2 Firmware Image with Authentication Support](image1.png)

![Figure 23-3 Firmware Image with Dependency/Authentication Support](image2.png)
Status Codes Returned

<table>
<thead>
<tr>
<th>Code</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>EFI_SUCCESS</td>
<td>The image information was successfully returned.</td>
</tr>
<tr>
<td>EFI_BUFFER_TOO_SMALL</td>
<td>The <code>ImageInfo</code> buffer was too small. The current buffer size needed to hold the image(s) information is returned in <code>ImageInfoSize</code>.</td>
</tr>
<tr>
<td>EFI_INVALID_PARAMETER</td>
<td><code>ImageInfoSize</code> is too small and <code>ImageInfo</code> is NULL.</td>
</tr>
<tr>
<td>EFI_INVALID_PARAMETER</td>
<td><code>ImageInfoSize</code> is too small and <code>DescriptorVersion</code> is NULL.</td>
</tr>
<tr>
<td>EFI_INVALID_PARAMETER</td>
<td><code>ImageInfoSize</code> is too small and <code>DescriptorCount</code> is NULL.</td>
</tr>
<tr>
<td>EFI_INVALID_PARAMETER</td>
<td><code>ImageInfoSize</code> is too small and <code>DescriptorSize</code> is NULL.</td>
</tr>
<tr>
<td>EFI_INVALID_PARAMETER</td>
<td><code>ImageInfoSize</code> is too small and <code>PackageVersion</code> is NULL.</td>
</tr>
<tr>
<td>EFI_INVALID_PARAMETER</td>
<td><code>ImageInfoSize</code> is too small and <code>PackageVersionName</code> is NULL.</td>
</tr>
<tr>
<td>EFI_DEVICE_ERROR</td>
<td>Valid information could not be returned. Possible corrupted image.</td>
</tr>
</tbody>
</table>

**EFI_FIRMWARE_MANAGEMENT_PROTOCOL.GetImage()**

**Summary**
Retrieves a copy of the current firmware image of the device.

**Protocol**

```c
typedef
EFI_STATUS
(EIFIAPI *EFI_FIRMWARE_MANAGEMENT_PROTOCOL_GET_IMAGE) (  
    IN EFI_FIRMWARE_MANAGEMENT_PROTOCOL *This,  
    IN UINT8 ImageIndex,  
    OUT VOID *Image,  
    IN OUT UINTN *ImageSize
);
```

**Parameters**

- **This**
  A pointer to the `EFI_FIRMWARE_MANAGEMENT_PROTOCOL` instance.

- **ImageIndex**
  A unique number identifying the firmware image(s) within the device. The number is between 1 and `DescriptorCount`.

- **Image**
  Points to the buffer where the current image is copied to. May be NULL with a zero `ImageSize` in order to determine the size of the buffer needed.

- **ImageSize**
  On entry, points to the size of the buffer pointed to by `Image`, in bytes. On return, points to the length of the image, in bytes.
Related Definitions
None

Description
This function allows a copy of the current firmware image to be created and saved. The saved copy could later be used, for example, in firmware image recovery or rollback.

Status Codes Returned

<table>
<thead>
<tr>
<th>Status Code</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>EFI_SUCCESS</td>
<td>The current image was successfully copied to the buffer.</td>
</tr>
<tr>
<td>EFI_BUFFER_TOO_SMALL</td>
<td>The buffer specified by ImageSize is too small to hold the image. The current buffer size needed to hold the image is returned in ImageSize.</td>
</tr>
<tr>
<td>EFI_INVALID_PARAMETER</td>
<td>The ImageSize is not too small and Image is NULL</td>
</tr>
<tr>
<td>EFI_NOT_FOUND</td>
<td>The current image is not copied to the buffer.</td>
</tr>
<tr>
<td>EFI_UNSUPPORTED</td>
<td>The operation is not supported.</td>
</tr>
<tr>
<td>EFI_SECURITY_VIOLATION</td>
<td>The operation could not be performed due to an authentication failure.</td>
</tr>
</tbody>
</table>

**EFI_FIRMWARE_MANAGEMENT_PROTOCOL.SetImage()**

Summary
Updates the firmware image of the device.

Protocol
```c
typedef EFI_STATUS
(EIFIAPI *EFI_FIRMWARE_MANAGEMENT_PROTOCOL_SET_IMAGE) (  
    IN EFI_FIRMWARE_MANAGEMENT_PROTOCOL *This,  
    IN UINT8 ImageIndex,  
    IN CONST VOID *Image,  
    IN UINTN ImageSize,  
    IN CONST VOID *VendorCode,  
    IN EFI_FIRMWARE_MANAGEMENT_UPDATE_IMAGE_PROGRESS Progress,  
    OUT CHAR16 **AbortReason
 ) ;
```

Parameters

This  
A pointer to the **EFI_FIRMWARE_MANAGEMENT_PROTOCOL** instance.

ImageIndex  
A unique number identifying the firmware image(s) within the device. The number is between 1 and DescriptorCount.

Image  
Points to the new image.
**ImageSize**

Size of the new image in bytes.

**VendorCode**

This enables vendor to implement vendor-specific firmware image update policy. Null indicates the caller did not specify the policy or use the default policy.

**Progress**

A function used by the driver to report the progress of the firmware update.

**AbortReason**

A pointer to a pointer to a null-terminated string providing more details for the aborted operation. The buffer is allocated by this function with AllocatePool(), and it is the caller’s responsibility to free it with a call to FreePool().

**Related Definitions**

```c
typedef
EFI_STATUS
(EFIAPI *EFI_FIRMWARE_MANAGEMENT_UPDATE_IMAGE_PROGRESS) (
    IN UINTN Completion
);
```

Completion

A value between 1 and 100 indicating the current completion progress of the firmware update. Completion progress is reported as from 1 to 100 percent. A value of 0 is used by the driver to indicate that progress reporting is not supported.

On EFI_SUCCESS, SetImage() continues to do the callback if supported. On NOT EFI_SUCCESS, SetImage() discontinues the callback and completes the update and returns.

**Description**

This function updates the hardware with the new firmware image.

This function returns EFI_UNSUPPORTED if the firmware image is not updatable.

If the firmware image is updatable, the function should perform the following minimal validations before proceeding to do the firmware image update.

- Validate the image authentication if image has attribute IMAGE_ATTRIBUTE_AUTHENTICATION_REQUIRED. The function returns EFI_SECURITY_VIOLATION if the validation fails.
- Validate the image is a supported image for this device. The function returns EFI_ABORTED if the image is unsupported. The function can optionally provide more detailed information on why the image is not a supported image.
- Validate the data from VendorCode if not null. Image validation must be performed before VendorCode data validation. VendorCode data is ignored or considered invalid if image validation failed. The function returns EFI_ABORTED if the data is invalid.
**VendorCode** enables vendor to implement vendor-specific firmware image update policy. Null if the caller did not specify the policy or use the default policy. As an example, vendor can implement a policy to allow an option to force a firmware image update when the abort reason is due to the new firmware image version is older than the current firmware image version or bad image checksum. Sensitive operations such as those wiping the entire firmware image and render the device to be non-functional should be encoded in the image itself rather than passed with the **VendorCode**.

**AbortReason** enables vendor to have the option to provide a more detailed description of the abort reason to the caller.

### Status Codes Returned

<table>
<thead>
<tr>
<th>Status Code</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>EFI_SUCCESS</td>
<td>The device was successfully updated with the new image.</td>
</tr>
<tr>
<td>EFI_ABORTED</td>
<td>The operation is aborted.</td>
</tr>
<tr>
<td>EFI_INVALID_PARAMETER</td>
<td>The Image was NULL.</td>
</tr>
<tr>
<td>EFI_UNSUPPORTED</td>
<td>The operation is not supported.</td>
</tr>
<tr>
<td>EFI_SECURITY_VIOLATION</td>
<td>The operation could not be performed due to an authentication failure.</td>
</tr>
</tbody>
</table>

#### EFI_FIRMWARE_MANAGEMENT_PROTOCOL.CheckImage()

**Summary**

Checks if the firmware image is valid for the device.

**Protocol**

```c
typedef EFI_STATUS
  (EFIAPI *EFI_FIRMWARE_MANAGEMENT_PROTOCOL_CHECK_IMAGE) (  
    IN EFI_FIRMWARE_MANAGEMENT_PROTOCOL *This,
    IN UINT8 ImageIndex,
    IN CONST VOID *Image,
    IN UINTN ImageSize,
    OUT UINT32 *ImageUpdatable
  ) ;
```

**Parameters**

- **This**
  A pointer to the **EFI_FIRMWARE_MANAGEMENT_PROTOCOL** instance.

- **ImageIndex**
  A unique number identifying the firmware image(s) within the device. The number is between 1 and **DescriptorCount**.

- **Image**
  Points to the new image.

- **ImageSize**
  Size of the new image in bytes.
ImageUpdatable

Indicates if the new image is valid for update. It also provides, if available, additional information if the image is invalid. See “Related Definitions”.

Related Definitions

//***************************************************************
// ImageUpdatable Definitions
//***************************************************************
#define IMAGE_UPDATABLE_VALID 0x0000000000000001
#define IMAGE_UPDATABLE_INVALID 0x0000000000000002
#define IMAGE_UPDATABLE_INVALID_TYPE 0x0000000000000004
#define IMAGE_UPDATABLE_INVALID_OLD 0x0000000000000008
#define IMAGE_UPDATABLE_VALID_WITH_VENDOR_CODE \ 0x0000000000000010

**IMAGE_UPDATABLE_VALID** indicates **setImage()** will accept the new image and update the device with the new image. The version of the new image could be higher or lower than the current image. **SetImage** VendorCode is optional but can be used for vendor specific action.

**IMAGE_UPDATABLE_INVALID** indicates **setImage()** will reject the new image. No additional information is provided for the rejection.

**IMAGE_UPDATABLE_INVALID_TYPE** indicates **setImage()** will reject the new image. The rejection is due to the new image is not a firmware image recognized for this device.

**IMAGE_UPDATABLE_INVALID_OLD** indicates **setImage()** will reject the new image. The rejection is due to the new image version is older than the current firmware image version in the device. The device firmware update policy does not support firmware version downgrade.

**IMAGE_UPDATABLE_VALID_WITH_VENDOR_CODE** indicates **setImage()** will accept and update the new image only if a correct VendorCode is provided or else image would be rejected and **SetImage** will return appropriate error.

**Description**

This function allows firmware update application to validate the firmware image without invoking the **SetImage()** first. Please see **SetImage()** for the type of image validations performed.
### Status Codes Returned

<table>
<thead>
<tr>
<th>Code</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>EFI_SUCCESS</td>
<td>The image was successfully checked.</td>
</tr>
<tr>
<td>EFI_INVALID_PARAMETER</td>
<td>The Image was NULL.</td>
</tr>
<tr>
<td>EFI_UNSUPPORTED</td>
<td>The operation is not supported.</td>
</tr>
<tr>
<td>EFI_SECURITY_VIOLATION</td>
<td>The operation could not be performed due to an authentication failure.</td>
</tr>
</tbody>
</table>

#### EFI_FIRMWARE_MANAGEMENT_PROTOCOL.GetPackageInfo()

**Summary**

Returns information about the firmware package.

**Protocol**

```c
typedef EFI_STATUS (EFIAPI *EFI_FIRMWARE_MANAGEMENT_PROTOCOL_GET_PACKAGE_INFO) (
    IN EFI_FIRMWARE_MANAGEMENT_PROTOCOL *This,
    OUT UINT32              *PackageVersion,
    OUT CHAR16 **PackageVersionName,
    OUT UINT32              PackageVersionNameMaxLen,
    OUT UINT64              *AttributesSupported,
    OUT UINT64              *AttributesSetting
);
```

**Parameters**

- **This**
  A pointer to the `EFI_FIRMWARE_MANAGEMENT_PROTOCOL` instance.

- **PackageVersion**
  A version number that represents all the firmware images in the device. The format is vendor specific and new version must have a greater value than the old version. If `PackageVersion` is not supported, the value is 0xFFFFFFFF. A value of 0xFFFFFFFF indicates that package version comparison is to be performed using `PackageVersionName`. A value of 0xFFFFFFFD indicates that package version update is in progress.

- **PackageVersionName**
  A pointer to a pointer to a null-terminated string representing the package version name. The buffer is allocated by this function with `AllocatePool()`, and it is the caller’s responsibility to free it with a call to `FreePool()`. 

- **PackageVersionNameMaxLen**
  The maximum length of package version name if device supports update of package version name. A value of 0 indicates the device does not support update of package version name.
version name. Length is the number of Unicode characters, including the terminating null character.

**AttributesSupported**

Package attributes that are supported by this device. See “Package Attribute Definitions” for possible returned values of this parameter. A value of 1 indicates the attribute is supported and the current setting value is indicated in **AttributesSetting**. A value of 0 indicates the attribute is not supported and the current setting value in **AttributesSetting** is meaningless.

**AttributesSetting**

Package attributes. See “Package Attribute Definitions” for possible returned values of this parameter.

**Related Definitions**

```c
//**************************************************************
// Package Attribute Definitions
//**************************************************************
#define PACKAGE_ATTRIBUTE_VERSION_UPDATABLE 0x0000000000000001
#define PACKAGE_ATTRIBUTE_RESET_REQUIRED   0x0000000000000002
#define PACKAGE_ATTRIBUTE_AUTHENTICATION_REQUIRED 0x0000000000000004
```

The attribute **PACKAGE_ATTRIBUTE_VERSION_UPDATABLE** indicates this device supports the update of the firmware package version.

The attribute **PACKAGE_ATTRIBUTE_RESET_REQUIRED** indicates a reset of the device is required for the new package info to take effect after an update.

The attribute **PACKAGE_ATTRIBUTE_AUTHENTICATION_REQUIRED** indicates authentication is required to update the package info.

**Description**

This function returns package information.

**Status Codes Returned**

<table>
<thead>
<tr>
<th>Code</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>EFI_SUCCESS</td>
<td>The package information was successfully returned.</td>
</tr>
<tr>
<td>EFI_UNSUPPORTED</td>
<td>The operation is not supported.</td>
</tr>
</tbody>
</table>
**Summary**

Updates information about the firmware package.

**Protocol**

```c
typedef EFI_STATUS (EFIAPI *EFI_FIRMWARE_MANAGEMENT_PROTOCOL_SET_PACKAGE_INFO) (
    IN EFI_FIRMWARE_MANAGEMENT_PROTOCOL  *This, 
    IN CONST VOID             *Image, 
    IN UINTN                   ImageSize, 
    IN CONST VOID             *VendorCode, 
    IN UINT32                  PackageVersion, 
    IN CONST CHAR16            *PackageVersionName 
) ;
```

**Parameters**

**This**

A pointer to the `EFI_FIRMWARE_MANAGEMENT_PROTOCOL` instance.

**Image**

Points to the authentication image. Null if authentication is not required.

**ImageSize**

Size of the authentication image in bytes. 0 if authentication is not required.

**VendorCode**

This enables vendor to implement vendor-specific firmware image update policy. Null indicates the caller did not specify this policy or use the default policy.

**PackageVersion**

The new package version.

**PackageVersionName**

A pointer to the new null-terminated Unicode string representing the package version name. The string length is equal to or less than the value returned in `PackageVersionNameMaxLen`.

**Description**

This function updates package information.

This function returns `EFI_UNSUPPORTED` if the package information is not updatable.

**VendorCode** enables vendor to implement vendor-specific package information update policy. Null if the caller did not specify this policy or use the default policy.
Status Codes Returned

<table>
<thead>
<tr>
<th>Code</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>EFI_SUCCESS</td>
<td>The device was successfully updated with the new package information</td>
</tr>
<tr>
<td>EFI_INVALID_PARAMETER</td>
<td>The <code>PackageVersionName</code> length is longer than the value returned in <code>PackageVersionNameMaxLen</code>.</td>
</tr>
<tr>
<td>EFI_UNSUPPORTED</td>
<td>The operation is not supported.</td>
</tr>
<tr>
<td>EFI_SECURITY_VIOLATION</td>
<td>The operation could not be performed due to an authentication failure.</td>
</tr>
</tbody>
</table>

23.2 Dependency Expression Instruction Set

The following topics describe each of the firmware management protocol dependency expression (depex) opcodes in detail. Information includes a description of the instruction functionality, binary encoding, and any limitations or unique behaviors of the instruction.

Several of the opcodes require a GUID operand. The GUID operand is a 16-byte value that matches the type `EFI_GUID` that is described in Chapter 2 of the UEFI 2.0 specification. These GUIDs represent the `EFI_FIRMWARE_IMAGE_DESCRIPTOR` `ImageTypeId` that are exposed by an `EFI_FIRMWARE_MANAGE_PROTOCOL` instance. A dependency expression is a packed byte stream of opcodes and operands. As a result, some of the GUID operands will not be aligned on natural boundaries. Care must be taken on processor architectures that do allow unaligned accesses.

The dependency expression is stored in a packed byte stream using postfix notation. As a dependency expression is evaluated, the operands are pushed onto a stack. Operands are popped off the stack to perform an operation. After the last operation is performed, the value on the top of the stack represents the evaluation of the entire dependency expression. If a push operation causes a stack overflow, then the entire dependency expression evaluates to `FALSE`. If a pop operation causes a stack underflow, then the entire dependency expression evaluates to `FALSE`. Reasonable implementations of a dependency expression evaluator should not make arbitrary assumptions about the maximum stack size it will support. Instead, it should be designed to grow the dependency expression stack as required. In addition, FMP images that contain dependency expressions should make an effort to keep their dependency expressions as small as possible to help reduce the size of the FMP image.

All opcodes are 8-bit values, and if an invalid opcode is encountered, then the entire dependency expression evaluates to `FALSE`.

When the dependency expression is being evaluated and a GUID specified cannot be found, then the result of the conditional operation evaluates to `FALSE`.

If, when evaluating two popped values from the stack, it is determined that they are of different types (e.g. BOOLEAN value and 32-bit value), then the entire dependency expression evaluates to `FALSE`.

If an END opcode is not present in a dependency expression, then the entire dependency expression evaluates to `FALSE`.

The final evaluation of the dependency expression results in either a `TRUE` or `FALSE` result.

Dependency Expression Opcode Summary

<table>
<thead>
<tr>
<th>Opcode</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x00</td>
<td>Push FMP GUID (1 op-code + 16 bytes)</td>
</tr>
<tr>
<td>0x01</td>
<td>Push 32-bit version value</td>
</tr>
</tbody>
</table>
PUSH_GUID

Syntax

PUSH_GUID <FMP GUID>

Description

Pushes the GUID value onto the stack. This GUID should be exposed by an EFI_FIRMWARE_MANAGEMENT_PROTOCOL instance. The GUID should match one of the EFI_FIRMWARE_IMAGE_DESCRIPTOR ImageTypeId values exposed through the GetImageInfo() function.

Operation

1) Search through all instances of the EFI_FIRMWARE_MANAGEMENT_PROTOCOL.

   a. In each instance, use the GetImageInfo() function to retrieve the ImageInfo->ImageTypeId value and ensure it matches the GUID specified in the op-code.

   b. If it doesn't match the GUID and no other instances match either, POP all values from the stack and PUSH FALSE onto the stack when evaluating a conditional operation involving the missing GUID.

2) Having found the matching EFI_FIRMWARE_MANAGEMENT_PROTOCOL instance, use the GetImageInfo() function and push the ImageInfo->Version value onto the stack.

Table 23-1 PUSH_GUID Instruction Encoding

<table>
<thead>
<tr>
<th>Byte</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x02</td>
<td>Declare NULL-terminated string (Human-readable Version)</td>
</tr>
<tr>
<td>0x03</td>
<td>AND – Pop 2 BOOLEAN values and Push TRUE if both are TRUE.</td>
</tr>
<tr>
<td>0x04</td>
<td>OR – Pop 2 BOOLEAN values and Push TRUE if either are TRUE.</td>
</tr>
<tr>
<td>0x05</td>
<td>NOT – Pop BOOLEAN value Push NOT of BOOLEAN value.</td>
</tr>
<tr>
<td>0x06</td>
<td>Push TRUE</td>
</tr>
<tr>
<td>0x07</td>
<td>Push FALSE</td>
</tr>
<tr>
<td>0x08</td>
<td>EQ – Pop 2 32-bit version values and push TRUE if equal.</td>
</tr>
<tr>
<td>0x09</td>
<td>GT - Pop 2 32-bit version values and push TRUE if first value is greater than the second.</td>
</tr>
<tr>
<td>0x0A</td>
<td>GTE - Pop 2 32-bit version values and push TRUE if first value is greater than or equal to the second.</td>
</tr>
<tr>
<td>0x0B</td>
<td>LT - Pop 2 32-bit version values and push TRUE if first value is less than the second.</td>
</tr>
<tr>
<td>0x0C</td>
<td>LTE - Pop 2 32-bit version values and push TRUE if first value is less than or equal to the second.</td>
</tr>
<tr>
<td>0x0D</td>
<td>END</td>
</tr>
<tr>
<td>0x0E</td>
<td>DECLARE_LENGTH - declares a 32-bit byte length of the entire dependency expression.</td>
</tr>
</tbody>
</table>
Behaviors and Restrictions
None.

**PUSH_VERSION**

**Syntax**

```
PUSH_VERSION <32-bit Version>
```

**Description**

Pushes the 32-bit version value to compare against onto the stack. This value will be used to compare against Version values exposed through the `GetImageInfo()` function.

**Table 23-2 PUSH_VERSION Instruction Encoding**

<table>
<thead>
<tr>
<th>Byte</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0x01</td>
</tr>
<tr>
<td>1..4</td>
<td>A 32-bit version to compare against.</td>
</tr>
</tbody>
</table>

Behaviors and Restrictions
None.

**DECLARE_VERSION_NAME**

**Syntax**

```
DECLARE_VERSION_NAME <NULL-terminated string>
```

**Description**

Declares an optional null-terminated version string that is the equivalent of the VersionName in the `EFI_FIRMWARE_MANAGEMENT_DESCRIPTOR`. Due to the OEM/IHV-specific format of version strings, this null-terminated string will not be used for purposes of comparison. Only the 32-bit integer values will be used for comparisons.

**Table 23-3 DECLARE_VERSION_NAME Instruction Encoding**

<table>
<thead>
<tr>
<th>Byte</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0x02</td>
</tr>
<tr>
<td>1..n</td>
<td>A null-terminated UNICODE string.</td>
</tr>
</tbody>
</table>

Behaviors and Restrictions
None.

**AND**

**Syntax**

---
AND

Description
Pops two Boolean operands off the stack, performs a Boolean AND operation between the two operands, and pushes the result back onto the stack.

Operation
Operand1 <= POP Boolean stack element
Operand2 <= POP Boolean stack element
Result <= Operand1 AND Operand2
PUSH Result

Table 23-4 AND Instruction Encoding

<table>
<thead>
<tr>
<th>Byte</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0x03</td>
</tr>
</tbody>
</table>

Behaviors and Restrictions
None.

OR

Syntax
OR

Description
Pops two Boolean operands off the stack, performs a Boolean OR operation between the two operands, and pushes the result back onto the stack.

Operation
Operand1 <= POP Boolean stack element
Operand2 <= POP Boolean stack element
Result <= Operand1 OR Operand2
PUSH Result

Table 23-5 OR Instruction Encoding

<table>
<thead>
<tr>
<th>Byte</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0x04</td>
</tr>
</tbody>
</table>

Behaviors and Restrictions
None.

NOT

Syntax
NOT

Description
Pops a Boolean operand off the stack, performs a Boolean NOT operation on the operand, and pushes the result back onto the stack.

Operation
\[
\text{Operand} \leftarrow \text{POP Boolean stack element} \\
\text{Result} \leftarrow \text{NOT Operand} \\
\text{PUSH Result}
\]

Table 23-6 NOT Instruction Encoding

<table>
<thead>
<tr>
<th>Byte</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0x05</td>
</tr>
</tbody>
</table>

Behaviors and Restrictions
None.

TRUE

Syntax
TRUE

Description
Pushes a Boolean TRUE onto the stack.

Operation
PUSH TRUE

Table 23-7 TRUE Instruction Encoding

<table>
<thead>
<tr>
<th>Byte</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0x06</td>
</tr>
</tbody>
</table>

Behaviors and Restrictions
None.

FALSE

Syntax
FALSE

Description
Pushes a Boolean FALSE onto the stack.

Operation
PUSH FALSE
Table 23-8 FALSE Instruction Encoding

<table>
<thead>
<tr>
<th>Byte</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0x07</td>
</tr>
</tbody>
</table>

Behaviors and Restrictions
None.

**EQ**

Syntax

```
EQ
```

Description
Pops two 32-bit operands off the stack, performs a Boolean equals comparison operation between the two operands, and pushes the result back onto the stack.

Operation

```
Operand1 ? POP 32-bit stack element
Operand2 ? POP 32-bit stack element
Result ? Operand1 EQ Operand2
PUSH Result
```

Table 23-9 EQ Instruction Encoding

<table>
<thead>
<tr>
<th>Byte</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0x08</td>
</tr>
</tbody>
</table>

Behaviors and Restrictions
None.

**GT**

Syntax

```
GT
```

Description
Pops two 32-bit operands off the stack, performs a Boolean greater-than comparison operation between the two operands, and pushes the result back onto the stack.

Operation

```
Operand1 <= POP 32-bit stack element
Operand2 <= POP 32-bit stack element
Result <= Operand1 GT Operand2
PUSH Result
```
Table 23-10 GT Instruction Encoding

<table>
<thead>
<tr>
<th>Byte</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0x09</td>
</tr>
</tbody>
</table>

Behaviors and Restrictions
None.

GTE

Syntax

GTE

Description
Pops two 32-bit operands off the stack, performs a Boolean greater-than-or-equal comparison operation between the two operands, and pushes the result back onto the stack.

Operation

\[
\begin{align*}
\text{Operand1} & \; \text{POP} \; \text{32-bit stack element} \\
\text{Operand2} & \; \text{POP} \; \text{32-bit stack element} \\
\text{Result} & \; \text{Operand1 GTE Operand2} \\
\text{PUSH Result} & 
\end{align*}
\]

Table 23-11 GTE Instruction Encoding

<table>
<thead>
<tr>
<th>Byte</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0x0A</td>
</tr>
</tbody>
</table>

Behaviors and Restrictions
None.

LT

Syntax

LT

Description
Pops two 32-bit operands off the stack, performs a Boolean less-than comparison operation between the two operands, and pushes the result back onto the stack.
Operation

Operand1 ? POP 32-bit stack element
Operand2 ? POP 32-bit stack element
Result ? Operand1 LT Operand2
PUSH Result

Table 23-12 LT Instruction Encoding

<table>
<thead>
<tr>
<th>Byte</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0x0B</td>
</tr>
</tbody>
</table>

Behaviors and Restrictions
None.

LTE

Syntax

LTE

Description
Pops two 32-bit operands off the stack, performs a Boolean less-than-or-equal comparison operation between the two operands, and pushes the result back onto the stack.

Operation

Operand1 ? POP 32-bit stack element
Operand2 ? POP 32-bit stack element
Result ? Operand1 LTE Operand2
PUSH Result

Table 23-13 LTE Instruction Encoding

<table>
<thead>
<tr>
<th>Byte</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0x0C</td>
</tr>
</tbody>
</table>

Behaviors and Restrictions
None.

END

Syntax

END

Description
Pops the final result of the dependency expression evaluation off the stack and exits the dependency expression evaluator.
Operation

POP Result
RETURN Result

Table 23-14 END Instruction Encoding

<table>
<thead>
<tr>
<th>Byte</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0x0D</td>
</tr>
</tbody>
</table>

Behaviors and Restrictions
This opcode must be the last one in a dependency expression.

---

**DECLARE_LENGTH**

Syntax

DECLARE_LENGTH <32-bit Length>

Description
Declares an 32-bit byte length of the entire dependency expression.

Table 23-15 DECLARE_LENGTH Instruction Encoding

<table>
<thead>
<tr>
<th>Byte</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0x0e</td>
</tr>
<tr>
<td></td>
<td>1..4 A 32-bit byte length for the entire dependency expression.</td>
</tr>
</tbody>
</table>

Behaviors and Restrictions
This opcode must be the first one in a dependency expression.
23.3 Delivering Capsules Containing Updates to Firmware Management Protocol

Summary
This section defines a method for delivery of a Firmware Management Protocol defined update using the UpdateCapsule runtime API.

23.3.1 EFI_FIRMWARE_MANAGEMENT_CAPSULE_ID_GUID

GUID

```
// {6DCBD5ED-E82D-4C44-BDA1-7194199AD92A}
#define EFI_FIRMWARE_MANAGEMENT_CAPSULE_ID_GUID
{0x6dcbd5ed, 0xe82d, 0x4c44, 
 {0xbd, 0xa1, 0x71, 0x94, 0x19, 0x9a, 0xd9, 0x2a }}
```

Description
This GUID is used in the CapsuleGuid field of EFI_Capsule_Header struct within a capsule constructed according to the definitions of section Section 8.5.3.1. Use of this GUID indicates a capsule with body conforming to the additional structure defined in Section 23.3.2.

When delivered to platform firmware QueryCapsuleCapabilities() the capsule will be examined according to the structure defined in Section 23.3.2 and if it is possible for the platform to process EFI_SUCCESS will be returned.

When delivered to platform firmware UpdateCapsule() the capsule will be examined according to the structure defined in Section 23.3.2 and if it is possible for the platform to process the update will be processed.

By definition Firmware Management protocol services are not available in EFI runtime and depending upon platform capabilities, EFI runtime delivery of this capsule may not be supported and may return an error when delivered in EFI runtime with CAPSULE_FLAGS_PERSIST_ACROSS_RESET bit defined. However any platform supporting this capability is required to accept this form of capsule in Boot Services, including optional use of CAPSULE_FLAGS_PERSIST_ACROSS_RESET bit.

23.3.2 DEFINED FIRMWARE MANAGEMENT PROTOCOL DATA CAPSULE STRUCTURE

Structure of the Capsule Body
Generic EFI Capsule Body is defined in Section 8.5.3.1. When an EFI Capsule is identified by EFI_FIRMWARE_MANAGEMENT_CAPSULE_ID_GUID, the internal structure of the capsule _FIRMWARE_MANAGEMENT_CAPSULE_HEADER followed by optional EFI drivers to be loaded by the platform and optional binary payload items to be processed and passed to Firmware Management Protocol image update function. Each binary payload item is preceded by EFI_FIRMWARE_MANAGEMENT_CAPSULE_IMAGE_HEADER. Internal capsule structure diagram follows.
Figure 23-4 Optional Scatter-Gather Construction of Capsule Submitted to UpdateCapsule()
Figure 23-5 Capsule Header and Firmware Management Capsule Header
Related Definitions

```c
#pragma pack(1)
typedef struct {
    UINT32 Version;
    UINT16 EmbeddedDriverCount;
    UINT16 PayloadItemCount;
    // UINT64 ItemOffsetList[];
} EFI_FIRMWARE_MANAGEMENT_CAPSULE_HEADER;
```

**Version**
Version of the structure, initially 0x00000001.

**EmbeddedDriverCount**
The number of drivers included in the capsule and the number of corresponding offsets stored in `ItemOffsetList` array. This field may be zero in the case where no driver is required.

**PayloadItemCount**
The number of payload items included in the capsule and the number of corresponding offsets stored in the `ItemOffsetList` array. This field may be zero in the case where no binary payload object is required to accomplish the update.
ItemOffsetList
Variable length array of dimension \([\text{EmbeddedDriverCount} + \text{PayloadItemCount}]\) containing offsets of each of the drivers and payload items contained within the capsule. The offsets of the items are calculated relative to the base address of the `EFI_FIRMWARE_MANAGEMENT_CAPSULE_HEADER` struct. Offset may indicate structure begins on any byte boundary. Offsets in the array must be sorted in ascending order with all drivers preceding all binary payload elements.

```c
#pragma pack(1)
typedef struct {
  UINT32   Version;
  EFI_GUID UpdateImageTypeId;
  UINT8    UpdateImageIndex;
  UINT8    reserved_bytes[3];
  UINT32   UpdateImageSize;
  UINT32   UpdateVendorCodeSize;
  UINT64   UpdateHardwareInstance;  //Introduced in v2
  UINT64   ImageCapsuleSupport;     //Introduced in v3
} EFI_FIRMWARE_MANAGEMENT_CAPSULE_IMAGE_HEADER;
```

Version
Version of the structure, initially \(0x00000003\).

UpdateImageTypeId
Used to identify device firmware targeted by this update. This guid is matched by system firmware against `ImageTypeId` field within a `EFI_FIRMWARE_IMAGE_DESCRIPTOR` returned by an instance of `EFI_FIRMWARE_MANAGEMENT_PROTOCOL.GetImageInfo()` in the system.

UpdateImageIndex
Passed as `ImageIndex` in call to `EFI_FIRMWARE_MANAGEMENT_PROTOCOL.SetImage()`.

UpdateImageSize
Size of the binary update image which immediately follows this structure. Passed as `ImageSize` to `EFI_FIRMWARE_MANAGEMENT_PROTOCOL.SetImage()`. This size may or may not include Firmware Image Authentication information.

UpdateVendorCodeSize
Size of the `VendorCode` bytes which optionally immediately follow binary update image in the capsule. Pointer to these bytes passed in `VendorCode` to `EFI_FIRMWARE_MANAGEMENT_PROTOCOL.SetImage()`. If `UpdateVendorCodeSize` is zero, then `VendorCode` is null in `SetImage()` call.
UpdateHardwareInstance
The HardwareInstance to target with this update. If value is zero it means match all HardwareInstances. This field allows update software to target only a single device in cases where there are more than one device with the same ImageTypeId GUID. This header is outside the signed data of the Authentication Info structure and therefore can be modified without changing the Auth data.

ImageCapsuleSupport
A 64-bit bitmask that determines what sections are added to the payload.
#define CAPSULE_SUPPORT_AUTHENTICATION 0x0000000000000001
#define CAPSULE_SUPPORT_DEPENDENCY 0x0000000000000002

Description
The EFI_FIRMWARE_MANAGEMENT_CAPSULE_HEADER structure is located at the lowest offset within the body of the capsule identified by EFI_FIRMWARE_MANAGEMENT_CAPSULE_ID_GUID. The structure is variable length with the number of element offsets within of the ItemOffsetList array determined by the count of drivers within the capsule plus the count of binary payload elements. It is expected that drivers whose presence is indicated by non-zero EmbeddedDriverCount will be used to supply an implementation of EFI_FIRMWARE_MANAGEMENT_PROTOCOL for devices that lack said protocol within the image to be updated.

Each payload item contained within the capsule body is preceded by a EFI_FIRMWARE_MANAGEMENT_CAPSULE_IMAGE_HEADER struct used to provide information required to prepare the payload item as an image for delivery to a instance of EFI_FIRMWARE_MANAGEMENT_PROTOCOL.SetImage() function.

Note: [Caution] The capsule identified by EFI_FIRMWARE_MANAGEMENT_CAPSULE_ID_GUID uses packed structures and structure fields may not be naturally aligned within the capsule buffer as delivered. Drivers and binary payload elements may start on byte boundary with no padding. Processing firmware may need to copy content elements during capsule unpacking in order to achieve any required natural alignment.

23.3.3 Firmware Processing of the Capsule Identified by EFI_FIRMWARE_MANAGEMENT_CAPSULE_ID_GUID
1. Capsule is presented to system firmware via call to UpdateCapsule() or using mass storage delivery procedure of Section 8.5.5. The capsule must be constructed to consist of a single EFI_FIRMWARE_MANAGEMENT_CAPSULE_HEADER structure with the 0 or more drivers and 0 or more binary payload items. However a capsule in which driver count and payload count are both zero is not processed.
2. Capsule is recognized by EFI_CAPSULE_HEADER member CapsuleGuid equal to EFI_FIRMWARE_MANAGEMENT_CAPSULE_ID_GUID. CAPSULE_FLAGS_POPULATE_SYSTEM_TABLE flag must be 0.
3. If system is not in boot services and platform does not support persistence of capsule across reset when initiated within EFI Runtime, EFI_OUT_OF_RESOURCES error is returned.
4. If device requires hardware reset to unlock flash write protection, CAPSULE_FLAGS_PERSIST ACROSS_RESET and optionally CAPSULE_FLAGS_INITIATE_RESET should be set to 1 in the EFI_CAPSULE_HEADER.
5. When reset is requested using `CAPSULE_FLAGS_PERSIST_ACROSS_RESET`, the capsule is processed in Boot Services, before the `EFI_EVENT_GROUP_READY_TO_BOOT` event.

6. All scatter-gather fragmentation is removed by the platform firmware and the capsule is processed as a contiguous buffer.

7. Examining `EFI_FIRMWARE_MANAGEMENT_CAPSULE_HEADER`, if `EmbeddedDriverCount` is non-zero, for each of the included drivers up to indicated count, the portion of the capsule body starting at the offset indicated by `ItemOffsetList[n]` and continuing for a size encompassing all bytes up to the next element’s offset stored in `ItemOffsetList[n+1]` or the end of the capsule, will be copied to a buffer. The driver contained within the capsule body may not be naturally aligned and the exact driver size in bytes should be respected to ensure successful security validation. In the case where a driver is last element in the `ItemOffsetList` array, the driver size may be calculated by reference to body size as calculated from `CapsuleImageSize` in `EFI_CAPSULE_HEADER`.

8. Each extracted driver is placed into a buffer and passed to `LoadImage()`. The driver image passed to `LoadImage()` must successfully pass all image format, platform type, and security checks including those related to UEFI secure boot, if enabled on the platform. After `LoadImage()` returns the processing of the capsule is continued with next driver if present until all drivers have been passed to `LoadImage()`. The driver being installed must check for matching hardware and instantiate any required protocols during call to `EFI_IMAGE_ENTRY_POINT`. In case where matching hardware is not found the driver should exit with error. In case where capsule creator has preference as to which of several included drivers to be made resident, later drivers in the capsule should confirm earlier driver successfully loaded and then exit with load error.

9. After driver processing is complete the platform firmware examines `PayloadItemCount`, and if zero the capsule processing is complete. Otherwise platform firmware sequentially locates each `EFI_FIRMWARE_MANAGEMENT_CAPSULE_IMAGE_HEADER` found within the capsule and processes according to steps 10-14.

10. For all instances of `EFI_FIRMWARE_MANAGEMENT_PROTOCOL` in the system, `GetImageInfo()` is called to return arrays of `EFI_FIRMWARE_IMAGE_DESCRIPTOR` structures.

11. Find the matching FMP instance(s):

   a. If the `EFI_FIRMWARE_MANAGEMENT_CAPSULE_IMAGE_HEADER` is version 1 or it is version 2 with `UpdateHardwareInstance` set to 0, then system firmware will use only the `ImageTypeId` to determine a match. For each instance of `EFI_FIRMWARE_MANAGEMENT_PROTOCOL` that returns a `EFI_FIRMWARE_IMAGE_DESCRIPTOR` containing an `ImageTypeId` GUID that matches the `UpdateImageTypeId` GUID within `EFI_FIRMWARE_MANAGEMENT_CAPSULE_IMAGE_HEADER`, the system firmware will call `SetImage()` function within that instance. In some cases there may be more than one instance of matching `EFI_FIRMWARE_MANAGEMENT_PROTOCOL` when multiple matching devices are installed in the system and all instances will be checked for GUID match and `SetImage()` call if match is successful.

   b. If the `EFI_FIRMWARE_MANAGEMENT_CAPSULE_IMAGE_HEADER` is version 2 and contains a non-zero value in the `UpdateHardwareInstance` field, then system firmware will use both `ImageTypeId` and `HardwareInstance` to determine a match. For the instance of `EFI_FIRMWARE_MANAGEMENT_PROTOCOL` that returns a `EFI_FIRMWARE_IMAGE_DESCRIPTOR` containing an `ImageTypeId` GUID that
matches the `UpdateImageTypeId` GUID and a `HardwareInstanceId` matching the `UpdateHardwareInstanceId` within `EFI_FIRMWARE.Management.Capsule.Image.Header`, the system firmware will call the `SetImage()` function within that instance. There will never be more than one instance since the `ImageId` must be unique.

12. In the situation where platform configuration or policy prohibits the processing of a capsule or individual FMP payload, the error `EFI_NOT_READY` will be returned in capsule result variable `CapsuleStatus` field. Otherwise `SetImage()` parameters are constructed using the `UpdateImageIndex`, `UpdateImageSize` and `UpdateVendorCodeSize` fields within `EFI_FIRMWARE.Management.Capsule.Image.Header`. In the case of capsule containing multiple payloads, or a payload matching multiple FMP instances, a separate Capsule Result Variable will be created with the results of each call to `SetImage()`. If any call to `SetImage()` selected per above matching algorithm returns an error, the processing of additional FMP instances or payload items in that capsule will be skipped and `EFI_ABORTED` returned in Capsule Result Variable for each potential call to `SetImage()` that was skipped.

13. `SetImage()` performs any required image authentication as described in that functions definition within this chapter.

14. Note: if multiple separate component updates including multiple `ImageIndex` values are required then additional `EFI_FIRMWARE.Management.Capsule.Image.Header` structures and image binaries are included within the capsule.

15. After all items in the capsule are processed the system is restarted by the platform firmware.

23.4 EFI System Resource Table

**EFI_SYSTEM_RESOURCE_TABLE**

**Summary**

The EFI System Resource Table (ESRT) provides an optional mechanism for identifying device and system firmware resources for the purposes of targeting firmware updates to those resources. Each entry in the ESRT describes a device or system firmware resource that can be targeted by a firmware capsule update. Each entry in the ESRT will also be used to report status of the last attempted update. See Section 4.6 for description of how to publish ESRT using `EFI_CONFIGURATION_TABLE`. The ESRT shall be stored in memory of type `EfiBootServicesData`. See Section 8.5.3 and Section 8.5.5 for details on delivery of updates to devices listed in ESRT.

**GUID**

```c
#define EFI_SYSTEMRESOURCE_TABLE_GUID
{ 0xb122a263, 0x3661, 0x4f68,
{ 0x99, 0x29, 0x78, 0xf8, 0xb0, 0xd6, 0x21, 0x80 }}
```

**Table Structure**

```c
typedef struct {
    UINT32       FwResourceCount;
    UINT32       FwResourceCountMax;
    UINT64       FwResourceVersion;
    //EFI_SYSTEMRESOURCEENTRY Entries[];
};
```
 Members

FwResourceCount  The number of firmware resources in the table, must not be zero.
FwResourceCountMax The maximum number of resource array entries that can be within the table without reallocating the table, must not be zero.
FwResourceVersion The version of the EFI_SYSTEM_RESOURCE_ENTRY entities used in this table. This field should be set to 1. See EFI_SYSTEM_RESOURCE_TABLE_FIRMWARE_RESOURCE_VERSION.

Entries  Array of EFI_SYSTEM_RESOURCE_ENTRY

Related Definitions

// Current Entry Version
#define EFI_SYSTEM_RESOURCE_TABLE_FIRMWARE_RESOURCE_VERSION 1

typedef struct {
    EFI_GUID  FwClass;
    UINT32    FwType;
    UINT32    FwVersion;
    UINT32    LowestSupportedFwVersion;
    UINT32    CapsuleFlags;
    UINT32    LastAttemptVersion;
    UINT32    LastAttemptStatus;
} EFI_SYSTEM_RESOURCE_ENTRY;

FwClass  The firmware class field contains a GUID that identifies a firmware component that can be updated via UpdateCapsule(). This GUID must be unique within all entries of the ESRT.
FwType  Identifies the type of firmware resource. See “Firmware Type Definitions” below for possible values.
FwVersion  The firmware version field represents the current version of the firmware resource, value must always increase as a larger number represents a newer version.
LowestSupportedFwVersion  The lowest firmware resource version to which a firmware resource can be rolled back for the given system/device. Generally this is used to protect against known and fixed security issues.
CapsuleFlags  The capsule flags field contains the CapsuleGuid flags (bits 0-15) as defined in the EFI_CAPSULE_HEADER that will be set in the capsule header.
LastAttemptVersion  The last attempt version field describes the last firmware version for which an update was attempted (uses the same format as Firmware Version).
Last Attempt Version is updated each time an `UpdateCapsule()` is attempted for an ESRT entry and is preserved across reboots (non-volatile). However, in cases where the attempt version is not recorded due to limitations in the update process, the field shall set to zero after a failed update. Similarly, in the case of a removable device, this value is set to 0 in cases where the device has not been updated since being added to the system.

### LastAttemptStatus

The last attempt status field describes the result of the last firmware update attempt for the firmware resource entry. `LastAttemptStatus` is updated each time an `UpdateCapsule()` is attempted for an ESRT entry and is preserved across reboots (non-volatile).

If a firmware update has never been attempted or is unknown, for example after fresh insertion of a removable device, `LastAttemptStatus` must be set to Success.

```c
// Firmware Type Definitions

#define ESRT_FW_TYPE_UNKNOWN         0x00000000
#define ESRT_FW_TYPE_SYSTEMFIRMWARE  0x00000001
#define ESRT_FW_TYPE_DEVICEFIRMWARE  0x00000002
#define ESRT_FW_TYPE_UEFIDRIVER      0x00000003

// Last Attempt Status Values

#define LAST_ATTEMPT_STATUS_SUCCESS                      0x00000000
#define LAST_ATTEMPT_STATUS_ERROR_UNSUCCESSFUL           0x00000001
#define LAST_ATTEMPT_STATUS_ERROR_INSUFFICIENT_RESOURCES 0x00000002
#define LAST_ATTEMPT_STATUS_ERROR_INCORRECT_VERSION      0x00000003
#define LAST_ATTEMPT_STATUS_ERROR_INVALID_FORMAT         0x00000004
#define LAST_ATTEMPT_STATUS_ERROR_AUTH_ERROR             0x00000005
#define LAST_ATTEMPT_STATUS_ERROR_PWR_EVT_AC             0x00000006
#define LAST_ATTEMPT_STATUS_ERROR_PWR_EVT_BATT           0x00000007
#define LAST_ATTEMPT_STATUS_ERROR_UNSATISFIED_DEPENDENCIES 0x00000008

// The LastAttemptStatus values of 0x1000 - 0x4000 are reserved for vendor usage.
#define LAST_ATTEMPT_STATUS_ERROR_UNSUCCESSFUL_VENDOR_RANGE_MIN 0x00001000
#define LAST_ATTEMPT_STATUS_ERROR_UNSUCCESSFUL_VENDOR_RANGE_MAX 0x00004000
```

### 23.4.1 Adding and Removing Devices from the ESRT

ESRT entries must be updated by System Firmware before handoff to the Operating System under the following conditions. Devices and systems that support hot swapping (once the OS has been loaded) will
not get their ESRT entries updated until the next reboot and execution of ESRT updating logic in the UEFI space.

**Required:** System firmware is responsible for updating the FirmwareVersion, LowestSupportedFirmwareVersion, LastAttemptVersion and LastAttemptStatus values in the ESRT any time UpdateCapsule is called and a firmware update attempt is made for the corresponding ESRT entry.

**Required:** the ESRT must be updated each time a configuration change is detected by system firmware, such as when a device is added or removed from the system.

**Optional:** all devices in the ESRT should be polled for any configuration changes any time UpdateCapsule is called.

### 23.4.2 ESRT and Firmware Management Protocol

Although the ESRT does not require firmware to use Firmware Management Protocol for updates it is designed to work with and extend the capabilities of FMP. The ESRT can be used to represent system and device firmware serviced by capsules that have an implementation specific format as well as devices that support Firmware Management Protocol and that are serviced by capsules formatted as described in Section 23.2, Delivering Capsules Containing Updates to Firmware Management Protocol. For system expansion devices, the task of building ESRT table entries is to be performed by the system firmware based upon FMP data published by the device.

### 23.4.3 Mapping Firmware Management Protocol Descriptors to ESRT Entries

Firmware Management Protocol descriptors define most of the information needed for an ESRT entry. The table below helps identify which members map to which fields. Some members are dependent on certain versions of FMP and it is left to system firmware to resolve any mappings when information is not present in the FMP instance. FMP descriptors should only be mapped to ESRT entries if the following are true:

- An entry with the same ImageTypeId is not already in the ESRT.
- AttributesSupported and AttributesSetting have the IMAGE_ATTRIBUTE_IN_USE bit set.
- In the case where DescriptorCount returned by GetImageInfo() is greater than one, firmware shall populate the ESRT according to system policy, noting however that multiple ESRT entries with identical FwClass values are not permitted.

<table>
<thead>
<tr>
<th>Table 23-16 ESRT and FMP Fields</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>ESRT Field</strong></td>
</tr>
<tr>
<td>FwClass</td>
</tr>
</tbody>
</table>
23.5 Delivering Capsule Containing JSON payload

Summary
This section defines a method for delivery of JSON payload to perform firmware configuration or firmware update using the UpdateCapsule runtime API or using mass storage delivery.

23.5.1 EFI_JSON_CAPSULE_ID_GUID

GUID

```
// {67D6F4CD-D6B8-4573-BF4A-DE5E252D61AE}
#define EFI_JSON_CAPSULE_ID_GUID
  {0x67d6f4cd, 0xd6b8, 0x4573, 
  {0xbf, 0x4a, 0xde, 0x5e, 0x25, 0x2d, 0x61, 0xae }}
```

Description
This GUID is used in the CapsuleGuid field of EFI_CAPSULE_HEADER struct within a capsule constructed according to the definitions of Section 8.5.3. Use of this GUID indicates a capsule with body conforming to the additional structure defined in Section 23.5.2.

When delivered to platform firmware QueryCapsuleCapabilities() the capsule will be examined according to the structure defined in Section 23.5.2, and if it is possible for the platform to process that then EFI_SUCCESS will be returned.

When delivered to platform firmware UpdateCapsule() the capsule will be examined according to the structure defined in Section 23.5.2, and if it is possible for the platform to process that the update will be processed.

By definition, firmware configuration and firmware update are not available in EFI runtime. Depending on platform capabilities, EFI runtime delivery of the capsule may not be supported, and may return an error when delivered in EFI runtime with CAPSULE_FLAGS_PERSIST_ACROSS_RESET bit defined. However, any platform supporting this capability is required to accept this form of capsule in Boot Services, including optional use of the CAPSULE_FLAGS_PERSIST_ACROSS_RESET bit.

<table>
<thead>
<tr>
<th>ESRT Field</th>
<th>FMP Field</th>
<th>Comment</th>
</tr>
</thead>
<tbody>
<tr>
<td>FwVersion</td>
<td>Version</td>
<td>Represents the current version of device firmware for an FMP instance.</td>
</tr>
<tr>
<td>LowestSupported FwVersion</td>
<td>LowestSupported ImageVersion</td>
<td>To be set after the completion of a firmware update attempt. In descriptor v3+ only. Default value is 0.</td>
</tr>
<tr>
<td>LastAttemptVersion</td>
<td>LastAttemptVersion</td>
<td>To be set after the completion of a firmware update attempt. In descriptor v3+ only. Default value is SUCCESS.</td>
</tr>
<tr>
<td>LastAttemptStatus</td>
<td>LastAttemptStatus</td>
<td>To be set after the completion of a firmware update attempt. In descriptor v3+ only. Default value is SUCCESS.</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>ESRT Field</th>
<th>FMP Field</th>
<th>Comment</th>
</tr>
</thead>
<tbody>
<tr>
<td>FwVersion</td>
<td>Version</td>
<td></td>
</tr>
<tr>
<td>LowestSupported FwVersion</td>
<td>LowestSupported ImageVersion</td>
<td></td>
</tr>
<tr>
<td>LastAttemptVersion</td>
<td>LastAttemptVersion</td>
<td></td>
</tr>
<tr>
<td>LastAttemptStatus</td>
<td>LastAttemptStatus</td>
<td></td>
</tr>
</tbody>
</table>
23.5.2 Defined JSON Capsule Data Structure

Structure of the Capsule Body
A generic EFI Capsule Body is defined in Section 8.5.3. When an EFI Capsule is identified by EFI_JSON_CAPSULE_ID_GUID, the internal structure of the capsule header is defined in this section, see EFI_JSON_CAPSULE_HEADER. Note that if multiple JSON capsules are delivered together, each JSON capsule should contain one EFI_CAPSULE_HEADER and one EFI_JSON_CAPSULE_HEADER separately.

Related Definitions

```c
#pragma pack(1)
typedef struct {
    UINT32 Version;
    UINT32 CapsuleId;
    UINT32 PayloadLength;
    UINT8 Payload[];
} EFI_JSON_CAPSULE_HEADER;
#pragma pack ()
```

Version
Version of the structure, initially 0x00000001.

CapsuleId
The unique identifier of this capsule.

PayloadLength
The length of the JSON payload immediately following this header, in bytes.

Payload
Variable length buffer containing the JSON payload that should be parsed and applied to the system. The definition of the JSON schema used in the payload is beyond the scope of this specification.

Description
The EFI_JSON_CAPSULE_HEADER structure is located at the lowest offset within the body of the capsule identified by EFI_JSON_CAPSULE_ID_GUID. It is expected that drivers which process the JSON payload have the specific knowledge of the JSON schema used in the payload. The drivers should parse the JSON payload firstly to understand whether the capsule wants to perform firmware configure or firmware update then route the JSON payload to corresponding modules. For instance, the capsule may be delivered to EFI_FIRMWARE_MANAGEMENT_PROTOCOL instance to update the firmware image.

Structure of the Configuration Data
During the system boot, current configuration data or cached configuration data is reported to the EFI System Configuration Table with EFI_JSON_CONFIG_DATA_TABLE_GUID according to the value of EFI_OS_INDICATIONS_JSON_CONFIG_DATA_REFRESH bit in OsIndications. The structure to record the configuration data is defined in this section, see EFI_JSON_CAPSULE_CONFIG_DATA.
Related Definitions

```c
#pragma pack(1)
typedef struct {
    UINT32 Version;
    UINT32 TotalLength;
    EFI_JSON_CONFIG_DATA_ITEM ConfigDataList[];
} EFI_JSON_CAPSULE_CONFIG_DATA;
#pragma pack ()
```

**Version**
Version of the structure, initially 0x00000001.

**TotalLength**
The total length of `EFI_JSON_CAPSULE_CONFIG_DATA`, in bytes.

**ConfigDataList**
Array of configuration data groups. Type `EFI_JSON_CONFIG_DATA_ITEM` is defined below.

```c
typedef struct {
    UINT32 ConfigDataLength;
    UINT8 ConfigData[];
} EFI_JSON_CONFIG_DATA_ITEM;
```

**ConfigDataLength**
The length of the following `ConfigData`, in bytes.

**ConfigData**
Variable length buffer containing the JSON payload that describes one group of configuration data within current system. The definition of the JSON schema used in this payload is beyond the scope of this specification.

**Description**
For supporting multiple groups of configuration data, a list of `EFI_JSON_CONFIG_DATA_ITEM` are included in `EFI_JSON_CAPSULE_CONFIG_DATA` and each item indicates one group of configuration data. It is expected that particular drivers have the specific knowledge of the JSON schema used in the payload so that they can describe system configuration data in JSON then install to the EFI System Configuration Table. The drivers should check `EFI_OS_INDICATIONS_JSON_CONFIG_DATA_REFRESH` bit in `OsIndications` to understand whether they need collect current configuration firstly.

### 23.5.3 Firmware Processing of the Capsule Identified by `EFI_JSON_CAPSULE_ID_GUID`

1. Capsule is presented to system firmware via call to `UpdateCapsule()` or using mass storage delivery procedure of Section 8.5.5. The capsule must be constructed to consist of a single `EFI_JSON_CAPSULE_HEADER` structure with JSON payload follows. A capsule in which `PayloadLength` is zero will not be processed.
2. Capsule is recognized by `EFI_CAPSULE_HEADER` member `CapsuleGuid` equal to `EFI_JSON_CAPSULE_ID_GUID`. `CAPSULE_FLAGS_POPULATE_SYSTEM_TABLE` flag must be 0.

3. If system is not in boot services and platform does not support persistence of capsule across reset when initiated within EFI Runtime, `EFI_OUT_OF_RESOURCES` error is returned.

4. If device requires hardware reset to unlock flash write protection, `CAPSULE_FLAGS_PERSIST_ACROSS_RESET` and optionally `CAPSULE_FLAGS_INITIATE_RESET` should be set to 1 in the `EFI_CAPSULE_HEADER`.

5. When reset is requested using `CAPSULE_FLAGS_PERSIST_ACROSS_RESET`, the capsule is processed in Boot Services, before the `EFI_EVENT_GROUP_READY_TO_BOOT` event.

6. All scatter-gather fragmentation is removed by the platform firmware and the capsule is processed as a contiguous buffer.

7. When a capsule identified by `EFI_JSON_CAPSULE_ID_GUID` is received, the system firmware shall place a pointer to the coalesced capsule in the EFI System Configuration Table with `EFI_JSON_CAPSULE_DATA_TABLE_GUID` before loading any third party modules such as option ROM. If multiple capsules identified by `EFI_JSON_CAPSULE_ID_GUID` are received, the system firmware shall place a list of pointers to the capsules, preceded by a UINTN that represents the number of pointers, in the EFI System Configuration Table with `EFI_JSON_CAPSULE_DATA_TABLE_GUID` before loading any third party modules such as option ROM. The UINTN and each pointer must be naturally aligned.

8. The system configuration driver should check EFI System Configuration Table and parse the JSON payload, to identify the configuration data type of JSON payload, and route the JSON payload to corresponding modules. The corresponding capsule pointer shall be removed from the EFI System Configuration Table and also be cleared after it is processed.

9. The processing result shall be installed to EFI System Configuration Table using the format of `EFI_CAPSULE_RESULT_VARIABLE_HEADER` and `EFI_CAPSULE_RESULT_VARIABLE_JSON` defined in Section 8.5.6 with `EFI_JSON_CAPSULE_RESULT_TABLE_GUID`. If the capsule is delivered via mass storage device, the process result shall be recorded by using UEFI variables as described in Section 8.5.6.
24.1 Simple Network Protocol

This section defines the Simple Network Protocol. This protocol provides a packet level interface to a network adapter.

**EFI_SIMPLE_NETWORK_PROTOCOL**

**Summary**

The **EFI_SIMPLE_NETWORK_PROTOCOL** provides services to initialize a network interface, transmit packets, receive packets, and close a network interface.

**GUID**

```
#define EFI_SIMPLE_NETWORK_PROTOCOL_GUID \
{0xA19832B9,0xAC25,0x11D3,\ 
 {0x9A,0x2D,0x00,0x90,0x27,0x3f,0xc1,0x4d}}
```

**Revision Number**

```
#define EFI_SIMPLE_NETWORK_PROTOCOL_REVISION 0x00010000
```

**Protocol Interface Structure**

```
typedef struct _EFI_SIMPLE_NETWORK_PROTOCOL_ {
    UINT64 Revision;
    EFI_SIMPLE_NETWORK_START Start;
    EFI_SIMPLE_NETWORK_STOP Stop;
    EFI_SIMPLE_NETWORK_INITIALIZE Initialize;
    EFI_SIMPLE_NETWORK_RESET Reset;
    EFI_SIMPLE_NETWORK_SHUTDOWN Shutdown;
    EFI_SIMPLE_NETWORK.ReceiveFilters ReceiveFilters;
    EFI_SIMPLE_NETWORK_STATION_ADDRESS StationAddress;
    EFI_SIMPLE_NETWORK_STATISTICS Statistics;
    EFI_SIMPLE_NETWORK_MCAST_IP_TO_MAC MCastIpToMac;
    EFI_SIMPLE_NETWORK_NVDATA NvData;
    EFI_SIMPLE_NETWORK_GET_STATUS GetStatus;
    EFI_SIMPLE_NETWORK_TRANSMIT Transmit;
    EFI_SIMPLE_NETWORK_RECEIVE Receive;
    EFI_EVENT WaitForPacket;
    *Mode;
} EFI_SIMPLE_NETWORK_PROTOCOL;
```

**Parameters**

- **Revision**
  
  Revision of the **EFI_SIMPLE_NETWORK_PROTOCOL**. All future revisions must be backwards compatible. If a future version is not backwards compatible it is not the same GUID.
<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Start</td>
<td>Prepares the network interface for further command operations. No other EFI_SIMPLE_NETWORK_PROTOCOL interface functions will operate until this call is made. See the Start() function description.</td>
</tr>
<tr>
<td>Stop</td>
<td>Stops further network interface command processing. No other EFI_SIMPLE_NETWORK_PROTOCOL interface functions will operate after this call is made until another Start() call is made. See the Stop() function description.</td>
</tr>
<tr>
<td>Initialize</td>
<td>Resets the network adapter and allocates the transmit and receive buffers. See the Initialize() function description.</td>
</tr>
<tr>
<td>Reset</td>
<td>Resets the network adapter and reinitializes it with the parameters provided in the previous call to Initialize(). See the Reset() function description.</td>
</tr>
<tr>
<td>Shutdown</td>
<td>Resets the network adapter and leaves it in a state safe for another driver to initialize. The memory buffers assigned in the Initialize() call are released. After this call, only the Initialize() or Stop() calls may be used. See the Shutdown() function description.</td>
</tr>
<tr>
<td>ReceiveFilters</td>
<td>Enables and disables the receive filters for the network interface and, if supported, manages the filtered multicast HW MAC (Hardware Media Access Control) address list. See the ReceiveFilters() function description.</td>
</tr>
<tr>
<td>StationAddress</td>
<td>Modifies or resets the current station address, if supported. See the StationAddress() function description.</td>
</tr>
<tr>
<td>Statistics</td>
<td>Collects statistics from the network interface and allows the statistics to be reset. See the Statistics() function description.</td>
</tr>
<tr>
<td>MCastIpToMac</td>
<td>Maps a multicast IP address to a multicast HW MAC address. See the MCastIPtoMAC() function description.</td>
</tr>
<tr>
<td>NvData</td>
<td>Reads and writes the contents of the NVRAM devices attached to the network interface. See the NvData() function description.</td>
</tr>
<tr>
<td>GetStatus</td>
<td>Reads the current interrupt status and the list of recycled transmit buffers from the network interface. See the GetStatus() function description.</td>
</tr>
<tr>
<td>Transmit</td>
<td>Places a packet in the transmit queue. See the Transmit() function description.</td>
</tr>
<tr>
<td>Receive</td>
<td>Retrieves a packet from the receive queue, along with the status flags that describe the packet type. See the Receive() function description.</td>
</tr>
<tr>
<td>WaitForPacket</td>
<td>Event used with EFI_BOOT_SERVICES.WaitForEvent() to wait for a packet to be received.</td>
</tr>
<tr>
<td>Mode</td>
<td>Pointer to the EFI_SIMPLE_NETWORK_MODE data for the device. See “Related Definitions” below.</td>
</tr>
</tbody>
</table>
Related Definitions

//**************************************************************
// EFI_SIMPLE_NETWORK_MODE
//**************************************************************
// Note that the fields in this data structure are read-only
// and are updated by the code that produces the
// EFI_SIMPLE_NETWORK_PROTOCOL
// functions. All these fields must be discovered
// in a protocol instance of
// EFI_DRIVER_BINDING_PROTOCOL.Start().
//**************************************************************
typedef struct {
    UINT32 State;
    UINT32 HwAddressSize;
    UINT32 MediaHeaderSize;
    UINT32 MaxPacketSize;
    UINT32 NvRamSize;
    UINT32 NvRamAccessSize;
    UINT32 ReceiveFilterMask;
    UINT32 ReceiveFilterSetting;
    UINT32 MaxMCastFilterCount;
    UINT32 MCastFilterCount;
    EFI_MAC_ADDRESS MCastFilter[MAX_MCAST_FILTER_CNT];
    EFI_MAC_ADDRESS CurrentAddress;
    EFI_MAC_ADDRESS BroadcastAddress;
    EFI_MAC_ADDRESS PermanentAddress;
    UINT8 IfType;
    BOOLEAN MacAddressChangeable;
    BOOLEAN MultipleTxSupported;
    BOOLEAN MediaPresentSupported;
    BOOLEAN MediaPresent;
} EFI_SIMPLE_NETWORK_MODE;

State          Reports the current state of the network interface (see
                EFI_SIMPLE_NETWORK_STATE below). When an
                EFI_SIMPLE_NETWORK_PROTOCOL driver initializes a network
                interface, the network interface is left in the
                EfiSimpleNetworkStopped state.
HwAddressSize  The size, in bytes, of the network interface’s HW address.
MediaHeaderSize The size, in bytes, of the network interface’s media header.
MaxPacketSize  The maximum size, in bytes, of the packets supported by the
                network interface.
NvRamSize      The size, in bytes, of the NVRAM device attached to the network
                interface. If an NVRAM device is not attached to the network
                interface, then this field will be zero. This value must be a multiple of
                NvramAccessSize.
NvRamAccessSize

The size that must be used for all NVRAM reads and writes. The start address for NVRAM read and write operations and the total length of those operations, must be a multiple of this value. The legal values for this field are 0, 1, 2, 4, and 8. If the value is zero, then no NVRAM devices are attached to the network interface.

ReceiveFilterMask

The multicast receive filter settings supported by the network interface.

ReceiveFilterSetting

The current multicast receive filter settings. See “Bit Mask Values for ReceiveFilterSetting” below.

MaxMCastFilterCount

The maximum number of multicast address receive filters supported by the driver. If this value is zero, then ReceiveFilters() cannot modify the multicast address receive filters. This field may be less than MAX_MCAST_FILTER_CNT (see below).

MCastFilterCount

The current number of multicast address receive filters.

MCastFilter

Array containing the addresses of the current multicast address receive filters.

CurrentAddress

The current HW MAC address for the network interface.

BroadcastAddress

The current HW MAC address for broadcast packets.

PermanentAddress

The permanent HW MAC address for the network interface.

IfType

The interface type of the network interface. See RFC 3232, section “Number Hardware Type.”

MacAddressChangeable

TRUE if the HW MAC address can be changed.

MultipleTxSupported

TRUE if the network interface can transmit more than one packet at a time.

MediaPresentSupported

TRUE if the presence of media can be determined; otherwise FALSE. If FALSE, MediaPresent cannot be used.

MediaPresent

TRUE if media are connected to the network interface; otherwise FALSE. This field shows the media present status as of the most recent GetStatus() call.
typedef enum {
    EfiSimpleNetworkStopped,
    EfiSimpleNetworkStarted,  
    EfiSimpleNetworkInitialized,
    EfiSimpleNetworkMaxState
} EFI_SIMPLE_NETWORK_STATE;

#define MAX_MCAST_FILTER_CNT 16

#define EFI_SIMPLE_NETWORK_RECEIVE_UNICAST 0x01
#define EFI_SIMPLE_NETWORK_RECEIVE_MULTICAST 0x02
#define EFI_SIMPLE_NETWORK_RECEIVE_BROADCAST 0x04
#define EFI_SIMPLE_NETWORK_RECEIVE_PROMISCUOUS 0x08
#define EFI_SIMPLE_NETWORK_RECEIVE_PROMISCUOUS_MULTICAST 0x10

Description
The EFI_SIMPLE_NETWORK_PROTOCOL protocol is used to initialize access to a network adapter. Once the network adapter initializes, the EFI_SIMPLE_NETWORK_PROTOCOL protocol provides services that allow packets to be transmitted and received. This provides a packet level interface that can then be used by higher level drivers to produce boot services like DHCP, TFTP, and MTFTP. In addition, this protocol can be used as a building block in a full UDP and TCP/IP implementation that can produce a wide variety of application level network interfaces. See the Preboot Execution Environment (PXE) Specification for more information.

Note: The underlying network hardware may only be able to access 4 GiB (32-bits) of system memory. Any requests to transfer data to/from memory above 4 GiB with 32-bit network hardware will be double-buffered (using intermediate buffers below 4 GiB) and will reduce performance.

Note: The same handle can have an instance of the EFI_ADAPTER_INFORMATION_PROTOCOL with a EFI_ADAPTER_INFO_MEDIA_STATE type structure.

EFI_SIMPLE_NETWORK.Start()

Summary
Changes the state of a network interface from “stopped” to “started.”
Prototype

```c
typedef
EFI_STATUS
(EIFIAPI *EFI_SIMPLE_NETWORK_START) (
    IN EFI_SIMPLE_NETWORK_PROTOCOL *This
);
```

Parameters
- `This` A pointer to the EFI_SIMPLE_NETWORK_PROTOCOL instance.

Description
This function starts a network interface. If the network interface successfully starts, then EFI_SUCCESS will be returned.

Status Codes Returned

<table>
<thead>
<tr>
<th>Code</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>EFI_SUCCESS</td>
<td>The network interface was started.</td>
</tr>
<tr>
<td>EFI_ALREADY_STARTED</td>
<td>The network interface is already in the started state.</td>
</tr>
<tr>
<td>EFI_INVALID_PARAMETER</td>
<td>This parameter was NULL or did not point to a valid EFI_SIMPLE_NETWORK_PROTOCOL structure.</td>
</tr>
<tr>
<td>EFI_DEVICE_ERROR</td>
<td>The command could not be sent to the network interface.</td>
</tr>
<tr>
<td>EFI_UNSUPPORTED</td>
<td>This function is not supported by the network interface.</td>
</tr>
</tbody>
</table>

EFI_SIMPLE_NETWORK.Stop()

Summary
Changes the state of a network interface from “started” to “stopped.”

Prototype

```c
typedef
EFI_STATUS
(EIFIAPI *EFI_SIMPLE_NETWORK_STOP) (
    IN EFI_SIMPLE_NETWORK_PROTOCOL *This
);
```

Parameters
- `This` A pointer to the EFI_SIMPLE_NETWORK_PROTOCOL instance.

Description
This function stops a network interface. This call is only valid if the network interface is in the started state. If the network interface was successfully stopped, then EFI_SUCCESS will be returned.
Status Codes Returned

<table>
<thead>
<tr>
<th>Code</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>EFI_SUCCESS</td>
<td>The network interface was stopped.</td>
</tr>
<tr>
<td>EFI_NOT_STARTED</td>
<td>The network interface has not been started.</td>
</tr>
<tr>
<td>EFI_INVALID_PARAMETER</td>
<td>This parameter was NULL or did not point to a valid EFI_SIMPLE_NETWORK_PROTOCOL structure.</td>
</tr>
<tr>
<td>EFI_DEVICE_ERROR</td>
<td>The command could not be sent to the network interface.</td>
</tr>
<tr>
<td>EFI_UNSUPPORTED</td>
<td>This function is not supported by the network interface.</td>
</tr>
</tbody>
</table>

EFI_SIMPLE_NETWORK.Initialize()

Summary
Resets a network adapter and allocates the transmit and receive buffers required by the network interface; optionally, also requests allocation of additional transmit and receive buffers.

Prototype
```c
typedef EFI_STATUS (EFIAPI *EFI_SIMPLE_NETWORK_INITIALIZE) (  
    IN EFI_SIMPLE_NETWORK_PROTOCOL *This,
    IN UINTN ExtraRxBufferSize OPTIONAL,
    IN UINTN ExtraTxBufferSize OPTIONAL
);
```

Parameters
- **This**
  A pointer to the EFI_SIMPLE_NETWORK_PROTOCOL instance.
- **ExtraRxBufferSize**
  The size, in bytes, of the extra receive buffer space that the driver should allocate for the network interface. Some network interfaces will not be able to use the extra buffer, and the caller will not know if it is actually being used.
- **ExtraTxBufferSize**
  The size, in bytes, of the extra transmit buffer space that the driver should allocate for the network interface. Some network interfaces will not be able to use the extra buffer, and the caller will not know if it is actually being used.

Description
This function allocates the transmit and receive buffers required by the network interface. If this allocation fails, then EFI_OUT_OF_RESOURCES is returned. If the allocation succeeds and the network interface is successfully initialized, then EFI_SUCCESS will be returned.
Status Codes Returned

<table>
<thead>
<tr>
<th>Code</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>EFI_SUCCESS</td>
<td>The network interface was initialized.</td>
</tr>
<tr>
<td>EFI_NOT_STARTED</td>
<td>The network interface has not been started.</td>
</tr>
<tr>
<td>EFI_OUT_OF_RESOURCES</td>
<td>There was not enough memory for the transmit and receive buffers.</td>
</tr>
<tr>
<td>EFI_INVALID_PARAMETER</td>
<td>This parameter was NULL or did not point to a valid EFI_SIMPLE_NETWORK_PROTOCOL structure.</td>
</tr>
<tr>
<td>EFI_DEVICE_ERROR</td>
<td>The command could not be sent to the network interface.</td>
</tr>
<tr>
<td>EFI_UNSUPPORTED</td>
<td>The increased buffer size feature is not supported.</td>
</tr>
</tbody>
</table>

EFI_SIMPLE_NETWORK.Reset()

Summary
Resets a network adapter and reinitializes it with the parameters that were provided in the previous call to Initialize().

Prototype
```c
typedef EFI_STATUS (EFIAPI *EFI_SIMPLE_NETWORK_RESET) (  
    IN EFI_SIMPLE_NETWORK_PROTOCOL  *This,  
    IN BOOLEAN          ExtendedVerification)
```

Parameters
- **This**: A pointer to the EFI_SIMPLE_NETWORK_PROTOCOL instance.
- **ExtendedVerification**: Indicates that the driver may perform a more exhaustive verification operation of the device during reset.

Description
This function resets a network adapter and reinitializes it with the parameters that were provided in the previous call to Initialize(). The transmit and receive queues are emptied and all pending interrupts are cleared. Receive filters, the station address, the statistics, and the multicast-IP-to-HW MAC addresses are not reset by this call. If the network interface was successfully reset, then EFI_SUCCESS will be returned. If the driver has not been initialized, EFI_DEVICE_ERROR will be returned.
Status Codes Returned

<table>
<thead>
<tr>
<th>Status Code</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>EFI_SUCCESS</td>
<td>The network interface was reset.</td>
</tr>
<tr>
<td>EFI_NOT_STARTED</td>
<td>The network interface has not been started.</td>
</tr>
<tr>
<td>EFI_INVALID_PARAMETER</td>
<td>One or more of the parameters has an unsupported value.</td>
</tr>
<tr>
<td>EFI_DEVICE_ERROR</td>
<td>The command could not be sent to the network interface.</td>
</tr>
<tr>
<td>EFI_UNSUPPORTED</td>
<td>This function is not supported by the network interface.</td>
</tr>
</tbody>
</table>

EFI_SIMPLE_NETWORK.Shutdown()

Summary
Resets a network adapter and leaves it in a state that is safe for another driver to initialize.

Prototype

typedef

EFI_STATUS

(EIFIAPI *EFI_SIMPLE_NETWORK_SHUTDOWN) ( |

IN EFI_SIMPLE_NETWORK_PROTOCOL *This |

);

Parameters

This A pointer to the EFI_SIMPLE_NETWORK_PROTOCOL instance.

Description
This function releases the memory buffers assigned in the Initialize() call. Pending transmits and receives are lost, and interrupts are cleared and disabled. After this call, only the Initialize() and Stop() calls may be used. If the network interface was successfully shutdown, then EFI_SUCCESS will be returned. If the driver has not been initialized, EFI_DEVICE_ERROR will be returned.

Status Codes Returned

<table>
<thead>
<tr>
<th>Status Code</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>EFI_SUCCESS</td>
<td>The network interface was shutdown.</td>
</tr>
<tr>
<td>EFI_NOT_STARTED</td>
<td>The network interface has not been started.</td>
</tr>
<tr>
<td>EFI_INVALID_PARAMETER</td>
<td>This parameter was NULL or did not point to a valid EFI_SIMPLE_NETWORK_PROTOCOL structure.</td>
</tr>
<tr>
<td>EFI_DEVICE_ERROR</td>
<td>The command could not be sent to the network interface.</td>
</tr>
</tbody>
</table>

EFI_SIMPLE_NETWORK.ReceiveFilters()

Summary
Manages the multicast receive filters of a network interface.
Prototype

```c
typedef
EFI_STATUS
(EFI_API *EFI_SIMPLE_NETWORK_RECEIVE_FILTERS) (
    IN EFI_SIMPLE_NETWORK_PROTOCOL  *This,
    IN UINT32           Enable,
    IN UINT32           Disable,
    IN BOOLEAN          ResetMCastFilter,
    IN UINTN            MCastFilterCnt OPTIONAL,
    IN EFI_MAC_ADDRESS  *MCastFilter OPTIONAL,
);```

Parameters

- **This**: A pointer to the `EFI_SIMPLE_NETWORK_PROTOCOL` instance.
- **Enable**: A bit mask of receive filters to enable on the network interface.
- **Disable**: A bit mask of receive filters to disable on the network interface. For backward compatibility with EFI 1.1 platforms, the `EFI_SIMPLE_NETWORK_RECEIVE_MULTICAST` bit must be set when the `ResetMCastFilter` parameter is `TRUE`.
- **MCastFilterCnt**: Number of multicast HW MAC addresses in the new MCastFilter list. This value must be less than or equal to the `MCastFilterCnt` field of `EFI_SIMPLE_NETWORK_MODE`. This field is optional if `ResetMCastFilter` is `TRUE`.
- **MCastFilter**: A pointer to a list of new multicast receive filter HW MAC addresses. This list will replace any existing multicast HW MAC address list. This field is optional if `ResetMCastFilter` is `TRUE`.

Description

This function is used to enable and disable the hardware and software receive filters for the underlying network device.

The receive filter change is broken down into three steps:

- The filter mask bits that are set (ON) in the Enable parameter are added to the current receive filter settings.
- The filter mask bits that are set (ON) in the Disable parameter are subtracted from the updated receive filter settings.
- If the resulting receive filter setting is not supported by the hardware a more liberal setting is selected.

If the same bits are set in the Enable and Disable parameters, then the bits in the Disable parameter takes precedence.
If the `ResetMCastFilter` parameter is TRUE, then the multicast address list filter is disabled (irregardless of what other multicast bits are set in the Enable and Disable parameters). The `SNP->Mode->MCastFilterCount` field is set to zero. The `Snp->Mode->MCastFilter` contents are undefined.

After enabling or disabling receive filter settings, software should verify the new settings by checking the `Snp->Mode->ReceiveFilterSettings`, `Snp->Mode->MCastFilterCount` and `Snp->Mode->MCastFilter` fields.

**Note:** Some network drivers and/or devices will automatically promote receive filter settings if the requested setting can not be honored. For example, if a request for four multicast addresses is made and the underlying hardware only supports two multicast addresses the driver might set the promiscuous or promiscuous multicast receive filters instead. The receiving software is responsible for discarding any extra packets that get through the hardware receive filters.

**Note:** To disable all receive filter hardware, the network driver must be `Shutdown()` and `Stopped()`. Calling `ReceiveFilters()` with `Disable` set to `Snp->Mode->ReceiveFilterSettings` will make it so no more packets are returned by the `Receive()` function, but the receive hardware may still be moving packets into system memory before inspecting and discarding them. Unexpected system errors, reboots and hangs can occur if an OS is loaded and the network devices are not `Shutdown()` and `Stopped()`.

If `ResetMCastFilter` is TRUE, then the multicast receive filter list on the network interface will be reset to the default multicast receive filter list. If `ResetMCastFilter` is FALSE, and this network interface allows the multicast receive filter list to be modified, then the `MCastFilterCnt` and `MCastFilter` are used to update the current multicast receive filter list. The modified receive filter list settings can be found in the `MCastFilter` field of `EFI_SIMPLE_NETWORK_MODE`. If the network interface does not allow the multicast receive filter list to be modified, then `EFI_INVALID_PARAMETER` will be returned. If the driver has not been initialized, `EFI_DEVICE_ERROR` will be returned.

If the receive filter mask and multicast receive filter list have been successfully updated on the network interface, `EFI_SUCCESS` will be returned.
Status Codes Returned

<table>
<thead>
<tr>
<th>Code</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>EFI_SUCCESS</td>
<td>The multicast receive filter list was updated.</td>
</tr>
<tr>
<td>EFI_NOT_STARTED</td>
<td>The network interface has not been started.</td>
</tr>
<tr>
<td>EFI_INVALID_PARAMETER</td>
<td>• One or more of the following conditions is <strong>TRUE</strong>:</td>
</tr>
<tr>
<td></td>
<td>• <strong>This</strong> is <strong>NULL</strong></td>
</tr>
<tr>
<td></td>
<td>• There are bits set in Enable that are not set in Snp-&gt;Mode- &gt;ReceiveFilterMask</td>
</tr>
<tr>
<td></td>
<td>• There are bits set in Disable that are not set in Snp-&gt;Mode- &gt;ReceiveFilterMask</td>
</tr>
<tr>
<td></td>
<td>• Multicast is being enabled (the EFI_SIMPLE_NETWORK_RECEIVE_MULTICAST bit is set in Enable, it is not set in Disable, and ResetMCastFilter is <strong>FALSE</strong>) and MCastFilterCount is zero</td>
</tr>
<tr>
<td></td>
<td>• Multicast is being enabled and MCastFilterCount is greater than Snp- &gt;Mode-&gt;MaxMCastFilterCount</td>
</tr>
<tr>
<td></td>
<td>• Multicast is being enabled and MCastFilter is <strong>NULL</strong></td>
</tr>
<tr>
<td></td>
<td>• Multicast is being enabled and one or more of the addresses in the MCastFilter list are not valid multicast MAC addresses</td>
</tr>
<tr>
<td>EFI_DEVICE_ERROR</td>
<td>• One or more of the following conditions is <strong>TRUE</strong>:</td>
</tr>
<tr>
<td></td>
<td>• The network interface has been started but has not been initialized</td>
</tr>
<tr>
<td></td>
<td>• An unexpected error was returned by the underlying network driver or device</td>
</tr>
<tr>
<td>EFI_UNSUPPORTED</td>
<td>This function is not supported by the network interface.</td>
</tr>
</tbody>
</table>

**EFI_SIMPLE_NETWORK.StationAddress()**

**Summary**

Modifies or resets the current station address, if supported.

**Prototype**

```c
typedef
EFI_STATUS
(EFIAPI *EFI_SIMPLE_NETWORK_STATION_ADDRESS) (  
  IN EFI_SIMPLE_NETWORK_PROTOCOL *This,  
  IN BOOLEAN Reset,  
  IN EFI_MAC_ADDRESS *New OPTIONAL  
);
```

**Parameters**

- **This**  
  A pointer to the EFI_SIMPLE_NETWORK_PROTOCOL instance.

- **Reset**  
  Flag used to reset the station address to the network interface’s permanent address.

- **New**  
  New station address to be used for the network interface.
Description
This function modifies or resets the current station address of a network interface, if supported. If `Reset` is `TRUE`, then the current station address is set to the network interface’s permanent address. If `Reset` is `FALSE`, and the network interface allows its station address to be modified, then the current station address is changed to the address specified by `New`. If the network interface does not allow its station address to be modified, then `EFI_INVALID_PARAMETER` will be returned. If the station address is successfully updated on the network interface, `EFI_SUCCESS` will be returned. If the driver has not been initialized, `EFI_DEVICE_ERROR` will be returned.

Status Codes Returned

<table>
<thead>
<tr>
<th>Status Code</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>EFI_SUCCESS</td>
<td>The network interface’s station address was updated.</td>
</tr>
<tr>
<td>EFI_NOT_STARTED</td>
<td>The Simple Network Protocol interface has not been started by calling <code>Start()</code>.</td>
</tr>
<tr>
<td>EFI_INVALID_PARAMETER</td>
<td>The <code>New</code> station address was not accepted by the NIC.</td>
</tr>
<tr>
<td>EFI_INVALID_PARAMETER</td>
<td><code>Reset</code> is <code>FALSE</code> and <code>New</code> is <code>NULL</code>.</td>
</tr>
<tr>
<td>EFI_DEVICE_ERROR</td>
<td>The Simple Network Protocol interface has not been initialized by calling <code>Initialize()</code>.</td>
</tr>
<tr>
<td>EFI_DEVICE_ERROR</td>
<td>An error occurred attempting to set the new station address.</td>
</tr>
<tr>
<td>EFI_UNSUPPORTED</td>
<td>The NIC does not support changing the network interface’s station address.</td>
</tr>
</tbody>
</table>

`EFI_SIMPLE_NETWORK.Statistics()`

Summary
Resets or collects the statistics on a network interface.

Prototype

```c
typedef
EFI_STATUS
(EFIAPI *EFI_SIMPLE_NETWORK_STATISTICS) (
  IN EFI_SIMPLE_NETWORK_PROTOCOL *This,
  IN BOOLEAN             Reset,
  IN OUT UINTN           *StatisticsSize OPTIONAL,
  OUT EFI_NETWORK_STATISTICS *StatisticsTable OPTIONAL
);
```

Parameters

- **This**
  A pointer to the `EFI_SIMPLE_NETWORK_PROTOCOL` instance.
- **Reset**
  Set to `TRUE` to reset the statistics for the network interface.
- **StatisticsSize**
  On input the size, in bytes, of `StatisticsTable`. On output the size, in bytes, of the resulting table of statistics.
- **StatisticsTable**
  A pointer to the `EFI_NETWORK_STATISTICS` structure that contains the statistics. Type `EFI_NETWORK_STATISTICS` is defined in “Related Definitions” below.
Related Definitions

// *******************************************************************************
// EFI_NETWORK_STATISTICS
//
// Any statistic value that is –1 is not available
// on the device and is to be ignored.
// *******************************************************************************
typedef struct {
    UINT64 RxTotalFrames;
    UINT64 RxGoodFrames;
    UINT64 RxUndersizeFrames;
    UINT64 RxOversizeFrames;
    UINT64 RxDroppedFrames;
    UINT64 RxUnicastFrames;
    UINT64 RxBroadcastFrames;
    UINT64 RxMulticastFrames;
    UINT64 RxCrcErrorFrames;
    UINT64 RxTotalBytes;
    UINT64 TxTotalFrames;
    UINT64 TxGoodFrames;
    UINT64 TxUndersizeFrames;
    UINT64 TxOversizeFrames;
    UINT64 TxDroppedFrames;
    UINT64 TxUnicastFrames;
    UINT64 TxBroadcastFrames;
    UINT64 TxMulticastFrames;
    UINT64 TxCrcErrorFrames;
    UINT64 TxTotalBytes;
    UINT64 Collisions;
    UINT64 UnsupportedProtocol;
    UINT64 RxDuplicatedFrames;
    UINT64 RxDecryptErrorFrames;
    UINT64 TxErrorFrames;
    UINT64 TxRetryFrames;
} EFI_NETWORK_STATISTICS;

RxTotalFrames     Total number of frames received. Includes frames with errors and dropped frames.
RxGoodFrames      Number of valid frames received and copied into receive buffers.
RxUndersizeFrames Number of frames below the minimum length for the communications device.
RxOversizeFrames  Number of frames longer than the maximum length for the communications device.
RxDroppedFrames   Valid frames that were dropped because receive buffers were full.
RxUnicastFrames  Number of valid unicast frames received and not dropped.
RxBroadcastFrames Number of valid broadcast frames received and not dropped.
RxMulticastFrames Number of valid multicast frames received and not dropped.
RxCrcErrorFrames  Number of frames with CRC or alignment errors.
RxTotalBytes Total number of bytes received. Includes frames with errors and dropped frames.
TxTotalFrames Total number of frames transmitted. Includes frames with errors and dropped frames.
TxGoodFrames Number of valid frames transmitted and copied into receive buffers.
TxUndersizeFrames Number of frames below the minimum length for the media. This would be less than 64 for Ethernet.
TxOversizeFrames Number of frames longer than the maximum length for the media. This would be greater than 1500 for Ethernet.
TxDroppedFrames Valid frames that were dropped because receive buffers were full.
TxUnicastFrames Number of valid unicast frames transmitted and not dropped.
TxBroadcastFrames Number of valid broadcast frames transmitted and not dropped.
TxMulticastFrames Number of valid multicast frames transmitted and not dropped.
TxCrcErrorFrames Number of frames with CRC or alignment errors.
TxTotalBytes Total number of bytes transmitted. Includes frames with errors and dropped frames.
Collisions Number of collisions detected on this subnet.
UnsupportedProtocol Number of frames destined for unsupported protocol.
RxDuplicatedFrames Number of valid frames received that were duplicated.
RxDecryptErrorFrames Number of encrypted frames received that failed to decrypt.
TxErrorFrames Number of frames that failed to transmit after exceeding the retry limit.
TxRetryFrames Number of frames transmitted successfully after more than one attempt.

**Description**

This function resets or collects the statistics on a network interface. If the size of the statistics table specified by `StatisticsSize` is not big enough for all the statistics that are collected by the network interface, then a partial buffer of statistics is returned in `StatisticsTable`, `StatisticsSize` is set to the size required to collect all the available statistics, and `EFI_BUFFER_TOO_SMALL` is returned.

If `StatisticsSize` is big enough for all the statistics, then `StatisticsTable` will be filled, `StatisticsSize` will be set to the size of the returned `StatisticsTable` structure, and `EFI_SUCCESS` is returned. If the driver has not been initialized, `EFI_DEVICE_ERROR` will be returned.

If `Reset` is `FALSE`, and both `StatisticsSize` and `StatisticsTable` are `NULL`, then no operations will be performed, and `EFI_SUCCESS` will be returned.

If `Reset` is `TRUE`, then all of the supported statistics counters on this network interface will be reset to zero.
Status Codes Returned

<table>
<thead>
<tr>
<th>Code</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>EFI_SUCCESS</td>
<td>The requested operation succeeded.</td>
</tr>
<tr>
<td>EFI_NOT_STARTED</td>
<td>The Simple Network Protocol interface has not been started by calling <code>Start()</code>.</td>
</tr>
<tr>
<td>EFI_BUFFER_TOO_SMALL</td>
<td><code>StatisticsSize</code> is not <code>NULL</code> and <code>StatisticsTable</code> is <code>NULL</code>. The current buffer size that is needed to hold all the statistics is returned in <code>StatisticsSize</code>. A partial set of statistics is returned in <code>StatisticsTable</code>.</td>
</tr>
<tr>
<td>EFI_INVALID_PARAMETER</td>
<td><code>StatisticsSize</code> is <code>NULL</code> and <code>StatisticsTable</code> is not <code>NULL</code>. The current buffer size that is needed to hold all the statistics is returned in <code>StatisticsSize</code>. A partial set of statistics is returned in <code>StatisticsTable</code>.</td>
</tr>
<tr>
<td>EFI_DEVICE_ERROR</td>
<td>The Simple Network Protocol interface has not been initialized by calling <code>Initialize()</code>.</td>
</tr>
<tr>
<td>EFI_DEVICE_ERROR</td>
<td>An error was encountered collecting statistics from the NIC.</td>
</tr>
<tr>
<td>EFI_UNSUPPORTED</td>
<td>The NIC does not support collecting statistics from the network interface.</td>
</tr>
</tbody>
</table>

EFI_SIMPLE_NETWORK.MCastIPtoMAC()

Summary
Converts a multicast IP address to a multicast HW MAC address.

Prototype
```c
typedef EFI_STATUS
    (EFIAPI *EFI_SIMPLE_NETWORK_MCAST_IP_TO_MAC) (    
    IN EFI_SIMPLE_NETWORK_PROTOCOL *This,  
    IN BOOLEAN IPv6,  
    IN EFI_IP_ADDRESS *IP,    
    OUT EFI_MAC_ADDRESS *MAC  
    );
```

Parameters
- `This` A pointer to the EFI_SIMPLE_NETWORK_PROTOCOL instance.
- `IPv6` Set to `TRUE` if the multicast IP address is IPv6 [RFC 2460]. Set to `FALSE` if the multicast IP address is IPv4 [RFC 791].
- `IP` The multicast IP address that is to be converted to a multicast HW MAC address.
- `MAC` The multicast HW MAC address that is to be generated from `IP`.

Description
This function converts a multicast IP address to a multicast HW MAC address for all packet transactions. If the mapping is accepted, then `EFI_SUCCESS` will be returned.
Status Codes Returned

<table>
<thead>
<tr>
<th>Code</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>EFI_SUCCESS</td>
<td>The multicast IP address was mapped to the multicast HW MAC address.</td>
</tr>
<tr>
<td>EFI_NOT_STARTED</td>
<td>The Simple Network Protocol interface has not been started by calling \texttt{Start()}.</td>
</tr>
<tr>
<td>EFI_INVALID_PARAMETER</td>
<td>\texttt{IP} is \texttt{NULL}</td>
</tr>
<tr>
<td>EFI_INVALID_PARAMETER</td>
<td>\texttt{MAC} is \texttt{NULL}</td>
</tr>
<tr>
<td>EFI_INVALID_PARAMETER</td>
<td>\texttt{IP} does not point to a valid IPv4 or IPv6 multicast address.</td>
</tr>
<tr>
<td>EFI_DEVICE_ERROR</td>
<td>The Simple Network Protocol interface has not been initialized by calling \texttt{Initialize()}.</td>
</tr>
<tr>
<td>EFI_UNSUPPORTED</td>
<td>IPv6 is \texttt{TRUE} and the implementation does not support IPv6 multicast to MAC address conversion.</td>
</tr>
</tbody>
</table>

EFI\_SIMPLE\_NETWORK.NvData()

Summary
Performs read and write operations on the NVRAM device attached to a network interface.

Prototype

```c
typedef
EFI_STATUS
(EFIAPIT *EFI\_SIMPLE\_NETWORK\_NVDATA) (  
  IN EFI\_SIMPLE\_NETWORK\_PROTOCOL *This
  IN BOOLEAN ReadWrite,
  IN UINTN Offset,
  IN UINTN BufferSize,
  IN OUT VOID *Buffer
);
```

Parameters

- **This**: A pointer to the EFI\_SIMPLE\_NETWORK\_PROTOCOL instance.
- **ReadWrite**: \texttt{TRUE} for read operations, \texttt{FALSE} for write operations.
- **Offset**: Byte offset in the NVRAM device at which to start the read or write operation. This must be a multiple of NvRamAccessSize and less than NvRamSize. (See EFI\_SIMPLE\_NETWORK\_MODE)
- **BufferSize**: The number of bytes to read or write from the NVRAM device. This must also be a multiple of 2.
- **Buffer**: A pointer to the data buffer.

Description
This function performs read and write operations on the NVRAM device attached to a network interface. If **ReadWrite** is \texttt{TRUE}, a read operation is performed. If **ReadWrite** is \texttt{FALSE}, a write operation is performed.
Offset specifies the byte offset at which to start either operation. Offset must be a multiple of NvRamAccessSize, and it must have a value between zero and NvRamSize.

BufferSize specifies the length of the read or write operation. BufferSize must also be a multiple of NvRamAccessSize, and Offset + BufferSize must not exceed NvRamSize.

If any of the above conditions is not met, then EFI_INVALID_PARAMETER will be returned.

If all the conditions are met and the operation is “read,” the NVRAM device attached to the network interface will be read into Buffer and EFI_SUCCESS will be returned. If this is a write operation, the contents of Buffer will be used to update the contents of the NVRAM device attached to the network interface and EFI_SUCCESS will be returned.

<table>
<thead>
<tr>
<th>Status Codes Returned</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>EFI_SUCCESS</td>
<td>The NVRAM access was performed.</td>
</tr>
<tr>
<td>EFI_NOT_STARTED</td>
<td>The network interface has not been started.</td>
</tr>
</tbody>
</table>
| EFI_INVALID_PARAMETER | One or more of the following conditions is TRUE:  
  • The This parameter is NULL  
  • The This parameter does not point to a valid EFI_SIMPLE_NETWORK_PROTOCOL structure  
  • The Offset parameter is not a multiple of EFI_SIMPLE_NETWORK_MODE.NvRamAccessSize  
  • The Offset parameter is not less than EFI_SIMPLE_NETWORK_MODE.NvRamSize  
  • TheBufferSize parameter is not a multiple of EFI_SIMPLE_NETWORK_MODE.NvRamAccessSize  
  The Buffer parameter is NULL |
| EFI_DEVICE_ERROR      | The command could not be sent to the network interface. |
| EFI_UNSUPPORTED      | This function is not supported by the network interface. |

**EFI_SIMPLE_NETWORK.GetStatus()**

**Summary**

Reads the current interrupt status and recycled transmit buffer status from a network interface.

**Prototype**

```c
typedef EFI_STATUS  
(EIFIAPI *EFI_SIMPLE_NETWORK_GET_STATUS) ( 
  IN EFI_SIMPLE_NETWORK_PROTOCOL  *This, 
  OUT UINT32  *InterruptStatus  OPTIONAL, 
  OUT VOID  **TxBuf  OPTIONAL
 );
```

**Parameters**

This A pointer to the EFI_SIMPLE_NETWORK_PROTOCOL instance.
InterruptStatus  A pointer to the bit mask of the currently active interrupts (see “Related Definitions”). If this is NULL, the interrupt status will not be read from the device. If this is not NULL, the interrupt status will be read from the device. When the interrupt status is read, it will also be cleared. Clearing the transmit interrupt does not empty the recycled transmit buffer array.

TxBuf  Recycled transmit buffer address. The network interface will not transmit if its internal recycled transmit buffer array is full. Reading the transmit buffer does not clear the transmit interrupt. If this is NULL, then the transmit buffer status will not be read. If there are no transmit buffers to recycle and TxBuf is not NULL, *TxBuf will be set to NULL.

Related Definitions

//**********************************************************************
// Interrupt Bit Mask Settings for InterruptStatus.
// Note that all other bit values are reserved.
//**********************************************************************
#define EFI_SIMPLE_NETWORK_RECEIVE_INTERRUPT     0x01
#define EFI_SIMPLE_NETWORK_TRANSMIT_INTERRUPT    0x02
#define EFI_SIMPLE_NETWORK_COMMAND_INTERRUPT     0x04
#define EFI_SIMPLE_NETWORK_SOFTWARE_INTERRUPT    0x08

Description

This function gets the current interrupt and recycled transmit buffer status from the network interface. The interrupt status is returned as a bit mask in InterruptStatus. If InterruptStatus is NULL, the interrupt status will not be read. Upon successful return of the media status, the MediaPresent field of EFI_SIMPLE_NETWORK_MODE will be updated to reflect any change of media status. Upon successful return of the media status, the MediaPresent field of EFI_SIMPLE_NETWORK_MODE will be updated to reflect any change of media status. If TxBuf is not NULL, a recycled transmit buffer address will be retrieved. If a recycled transmit buffer address is returned in TxBuf, then the buffer has been successfully transmitted, and the status for that buffer is cleared. If the status of the network interface is successfully collected, EFI_SUCCESS will be returned. If the driver has not been initialized, EFIDEVICE_ERROR will be returned.

Status Codes Returned

<table>
<thead>
<tr>
<th>Code</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>EFI_SUCCESS</td>
<td>The status of the network interface was retrieved.</td>
</tr>
<tr>
<td>EFI_NOT_STARTED</td>
<td>The network interface has not been started.</td>
</tr>
<tr>
<td>EFI_INVALID_PARAMETER</td>
<td>This parameter was NULL or did not point to a valid EFI_SIMPLE_NETWORK_PROTOCOL structure.</td>
</tr>
<tr>
<td>EFI_DEVICE_ERROR</td>
<td>The command could not be sent to the network interface.</td>
</tr>
</tbody>
</table>

EFI_SIMPLE_NETWORK.Transmit()

Summary

Places a packet in the transmit queue of a network interface.
Prototype

```c
typedef EFI_STATUS
(EIFIAPI *EFI_SIMPLE_NETWORK_TRANSMIT) (
    IN EFI_SIMPLE_NETWORK_PROTOCOL *This,
    IN UINTN HeaderSize,
    IN UINTN BufferSize,
    IN VOID *Buffer,
    IN EFI_MAC_ADDRESS *SrcAddr OPTIONAL,
    IN EFI_MAC_ADDRESS *DestAddr OPTIONAL,
    IN UINT16 *Protocol OPTIONAL,
);
```

Parameters

- **This**: A pointer to the EFI_SIMPLE_NETWORK_PROTOCOL instance.
- **HeaderSize**: The size, in bytes, of the media header to be filled in by the Transmit() function. If HeaderSize is nonzero, then it must be equal to This->Mode->MediaHeaderSize and the DestAddr and Protocol parameters must not be NULL.
- **BufferSize**: The size, in bytes, of the entire packet (media header and data) to be transmitted through the network interface.
- **Buffer**: A pointer to the packet (media header followed by data) to be transmitted. This parameter cannot be NULL. If HeaderSize is zero, then the media header in Buffer must already be filled in by the caller. If HeaderSize is nonzero, then the media header will be filled in by the Transmit() function.
- **SrcAddr**: The source HW MAC address. If HeaderSize is zero, then this parameter is ignored. If HeaderSize is nonzero and SrcAddr is NULL, then This->Mode->CurrentAddress is used for the source HW MAC address.
- **DestAddr**: The destination HW MAC address. If HeaderSize is zero, then this parameter is ignored.
- **Protocol**: The type of header to build. If HeaderSize is zero, then this parameter is ignored. See RFC 3232, section “Ether Types,” for examples.

Description

This function places the packet specified by **Header** and **Buffer** on the transmit queue. If **HeaderSize** is nonzero and **HeaderSize** is not equal to This->Mode->MediaHeaderSize, then **EFI_INVALID_PARAMETER** will be returned. If **BufferSize** is less than This->Mode->MediaHeaderSize, then **EFI_BUFFER_TOO_SMALL** will be returned. If Buffer is NULL, then **EFI_INVALID_PARAMETER** will be returned. If **HeaderSize** is nonzero and DestAddr or **Protocol** is NULL, then **EFI_INVALID_PARAMETER** will be returned. If the transmit engine of the network interface is busy, then **EFI_NOT_READY** will be returned. If this packet can be accepted by the transmit engine of the network interface, the packet contents specified by **Buffer** will be placed on the transmit queue of the network interface, and **EFI_SUCCESS** will be returned. **GetStatus()** can be
used to determine when the packet has actually been transmitted. The contents of the Buffer must not be modified until the packet has actually been transmitted.

The Transmit() function performs nonblocking I/O. A caller who wants to perform blocking I/O, should call Transmit(), and then GetStatus() until the transmitted buffer shows up in the recycled transmit buffer.

If the driver has not been initialized, EFI_DEVICE_ERROR will be returned.

**Status Codes Returned**

<table>
<thead>
<tr>
<th>Code</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>EFI_SUCCESS</td>
<td>The packet was placed on the transmit queue.</td>
</tr>
<tr>
<td>EFI_NOT_STARTED</td>
<td>The network interface has not been started.</td>
</tr>
<tr>
<td>EFI_NOT_READY</td>
<td>The network interface is too busy to accept this transmit request.</td>
</tr>
<tr>
<td>EFI_BUFFER_TOO_SMALL</td>
<td>The BufferSize parameter is too small.</td>
</tr>
<tr>
<td>EFI_INVALID_PARAMETER</td>
<td>One or more of the parameters has an unsupported value.</td>
</tr>
<tr>
<td>EFI_DEVICE_ERROR</td>
<td>The command could not be sent to the network interface.</td>
</tr>
<tr>
<td>EFI_UNSUPPORTED</td>
<td>This function is not supported by the network interface.</td>
</tr>
</tbody>
</table>

**EFI_SIMPLE_NETWORK.Receive()**

**Summary**
Receives a packet from a network interface.

**Prototype**

```c
typedef EFI_STATUS (EFIAPIC *EFI_SIMPLE_NETWORK_RECEIVE) (  
    IN EFI_SIMPLE_NETWORK_PROTOCOL *This,  
    OUT UINTN *HeaderSize OPTIONAL,  
    IN OUT UINTN *BufferSize,  
    OUT VOID *Buffer,  
    OUT EFI_MAC_ADDRESS *SrcAddr OPTIONAL,  
    OUT EFI_MAC_ADDRESS *DestAddr OPTIONAL,  
    OUT UINT16 *Protocol OPTIONAL  
);  
```

**Parameters**

- **This** A pointer to the EFI_SIMPLE_NETWORK_PROTOCOL instance.
- **HeaderSize** The size, in bytes, of the media header received on the network interface. If this parameter is NULL, then the media header size will not be returned.
- **BufferSize** On entry, the size, in bytes, of Buffer. On exit, the size, in bytes, of the packet that was received on the network interface.
- **Buffer** A pointer to the data buffer to receive both the media header and the data.
The source HW MAC address. If this parameter is NULL, the HW MAC source address will not be extracted from the media header.

DestAddr

The destination HW MAC address. If this parameter is NULL, the HW MAC destination address will not be extracted from the media header.

Protocol

The media header type. If this parameter is NULL, then the protocol will not be extracted from the media header. See RFC 1700 section “Ether Types” for examples.

Description

This function retrieves one packet from the receive queue of a network interface. If there are no packets on the receive queue, then EFI_NOT_READY will be returned. If there is a packet on the receive queue, and the size of the packet is smaller than BufferSize, then the contents of the packet will be placed in Buffer, and BufferSize will be updated with the actual size of the packet. In addition, if SrcAddr, DestAddr, and Protocol are not NULL, then these values will be extracted from the media header and returned. EFI_SUCCESS will be returned if a packet was successfully received. If BufferSize is smaller than the received packet, then the size of the receive packet will be placed in BufferSize and EFI_BUFFER_TOO_SMALL will be returned. If the driver has not been initialized, EFI_DEVICE_ERROR will be returned.

Status Codes Returned

<table>
<thead>
<tr>
<th>Code</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>EFI_SUCCESS</td>
<td>The received data was stored in Buffer, and BufferSize has been updated to the number of bytes received.</td>
</tr>
<tr>
<td>EFI_NOT_STARTED</td>
<td>The network interface has not been started.</td>
</tr>
<tr>
<td>EFI_NOT_READY</td>
<td>No packets have been received on the network interface.</td>
</tr>
<tr>
<td>EFI_BUFFER_TOO_SMALL</td>
<td>BufferSize is too small for the received packets. BufferSize has been updated to the required size.</td>
</tr>
<tr>
<td>EFI_INVALID_PARAMETER</td>
<td>One or more of the following conditions is TRUE:</td>
</tr>
<tr>
<td></td>
<td>• The This parameter is NULL</td>
</tr>
<tr>
<td></td>
<td>• The This parameter does not point to a valid EFI_SIMPLE_NETWORK_PROTOCOL structure.</td>
</tr>
<tr>
<td></td>
<td>• TheBufferSize parameter is NULL</td>
</tr>
<tr>
<td></td>
<td>• The Buffer parameter is NULL</td>
</tr>
<tr>
<td>EFI_DEVICE_ERROR</td>
<td>The command could not be sent to the network interface.</td>
</tr>
</tbody>
</table>

24.2 Network Interface Identifier Protocol

This is an optional protocol that is used to describe details about the software layer that is used to produce the Simple Network Protocol. This protocol is only required if the underlying network interface is 16-bit UNDI, 32/64-bit S/W UNDI, or H/W UNDI. It is used to obtain type and revision information about the underlying network interface.

An instance of the Network Interface Identifier protocol must be created for each physical external network interface that is controlled by the iPXE structure. The iPXE structure is defined in the 32/64-bit UNDI Specification in Appendix E.
**EFI_NETWORK_INTERFACE_IDENTIFIER_PROTOCOL**

**Summary**
An optional protocol that is used to describe details about the software layer that is used to produce the Simple Network Protocol.

**GUID**
```
#define EFI_NETWORK_INTERFACE_IDENTIFIER_PROTOCOL_GUID_31 \ 
{0x1ACED566, 0x76ED, 0x4218, \ 
{0xBC, 0x81, 0x76, 0x7F, 0x1F, 0x97, 0x7A, 0x89}}
```

**Revision Number**
```
#define EFI_NETWORK_INTERFACE_IDENTIFIER_PROTOCOL_REVISION \ 
0x00020000
```

**Protocol Interface Structure**
```
typedef struct {
    UINT64  Revision;
    UINT64  Id;
    UINT64  ImageAddr;
    UINT32  ImageSize;
    CHAR8   StringId[4];
    UINT8   Type;
    UINT8   MajorVer;
    UINT8   MinorVer;
    BOOLEAN Ipv6Supported;
    UINT16  IfNum;
} EFI_NETWORK_INTERFACE_IDENTIFIER_PROTOCOL;
```

**Parameters**
- **Revision**
The revision of the **EFI_NETWORK_INTERFACE_IDENTIFIER** protocol.
- **Id**
Address of the first byte of the identifying structure for this network interface. This is only valid when the network interface is started (see **Start()**). When the network interface is not started, this field is set to zero.
  - **16-bit UNDI and 32/64-bit S/W UNDI:**
    - `Id` contains the address of the first byte of the copy of the `!PXE` structure in the relocated UNDI code segment. See the **Preboot Execution Environment (PXE) Specification** and Appendix E.
  - **H/W UNDI:**
    - `Id` contains the address of the **!PXE** structure.
- **ImageAddr**
Address of the unrelocated network interface image.
  - **16-bit UNDI:**
    - `ImageAddr` is the address of the PXE option ROM image in upper memory.
32/64-bit S/W UNDI:

- **ImageAddr** is the address of the unrelocated S/W UNDI image.

H/W UNDI:

- **ImageAddr** contains zero.

- **ImageSize** is the size of the network interface image.

16-bit UNDI:

- **ImageSize** is the size of the PXE option ROM image in upper memory.

32/64-bit S/W UNDI:

- **ImageSize** is the size of the unrelocated S/W UNDI image.

H/W UNDI:

- **ImageSize** contains zero.

- **StringId** is a four-character ASCII string that is sent in the class identifier field of option 60 in DHCP. For a **Type** of **EfiNetworkInterfaceUndi**, this field is “UNDI.”

- **Type** is the network interface type. This will be set to one of the values in **EFI_NETWORK_INTERFACE_TYPE** (see “Related Definitions” below).

- **MajorVer** is the major version number.

16-bit UNDI:

- **MajorVer** comes from the third byte of the **UNDIRev** field in the **UNDI ROM ID** structure. Refer to the **Preboot Execution Environment (PXE) Specification**.

32/64-bit S/W UNDI and H/W UNDI:

- **MajorVer** comes from the **Major** field in the **!PXE** structure. See Appendix E.

- **MinorVer** is the minor version number.

16-bit UNDI:

- **MinorVer** comes from the second byte of the **UNDIRev** field in the **UNDI ROM ID** structure. Refer to the **Preboot Execution Environment (PXE) Specification**.

32/64-bit S/W UNDI and H/W UNDI:

- **MinorVer** comes from the **Minor** field in the **!PXE** structure. See Appendix E.

- **Ipv6Supported** is **TRUE** if the network interface supports IPv6; otherwise **FALSE**.

- **IfNum** is the network interface number that is being identified by this Network Interface Identifier Protocol. This field must be less than or equal to the (IFcnt | IFcntExt << 8) field in the **!PXE** structure.
Related Definitions

```c
typedef enum {
    EfiNetworkInterfaceUndi = 1
} EFI_NETWORK_INTERFACE_TYPE;
```

Description

The `EFI_NETWORK_INTERFACE_IDENTIFIER_PROTOCOL` is used by `EFI_PXE_BASE_CODE_PROTOCOL` and OS loaders to identify the type of the underlying network interface and to locate its initial entry point.

24.3 PXE Base Code Protocol

This section defines the Preboot Execution Environment (PXE) Base Code protocol, which is used to access PXE-compatible devices for network access and network booting. For more information about PXE, see “Links to UEFI-Related Documents” ([http://uefi.org/uefi](http://uefi.org/uefi)) under the heading “Preboot Execution Environment (PXE) Specification”.

**EFI_PXE_BASE_CODE_PROTOCOL**

**Summary**

The `EFI_PXE_BASE_CODE_PROTOCOL` is used to control PXE-compatible devices. The features of these devices are defined in the *Preboot Execution Environment (PXE) Specification*. An `EFI_PXE_BASE_CODE_PROTOCOL` will be layered on top of an `EFI_MANAGED_NETWORK_PROTOCOL` protocol in order to perform packet level transactions. The `EFI_PXE_BASE_CODE_PROTOCOL` handle also supports the `EFI_LOAD_FILE_PROTOCOL` protocol. This provides a clean way to obtain control from the boot manager if the boot path is from the remote device.

**GUID**

```c
#define EFI_PXE_BASE_CODE_PROTOCOL_GUID \\
{0x03C4E603,0xAC28,0x11d3,\\
{0x9A,0x2D,0x00,0x90,0x27,0x3F,0xC1,0x4D}}
```

**Revision Number**

```c
#define EFI_PXE_BASE_CODE_PROTOCOL_REVISION 0x00010000
```
Protocol Interface Structure

typedef struct {
  UINT64 Revision;
  EFI_PXE_BASE_CODE_START Start;
  EFI_PXE_BASE_CODE_STOP Stop;
  EFI_PXE_BASE_CODE_DHCP Dhcp;
  EFI_PXE_BASE_CODE_DISCOVER Discover;
  EFI_PXE_BASE_CODE_MTFTP Mtftp;
  EFI_PXE_BASE_CODE_UDP_WRITE UdpWrite;
  EFI_PXE_BASE_CODE_UDP_READ UdpRead;
  EFI_PXE_BASE_CODE_SET_IP_FILTER SetIpFilter;
  EFI_PXE_BASE_CODE_ARP Arp;
  EFI_PXE_BASE_CODE_SET_PARAMETERS SetParameters;
  EFI_PXE_BASE_CODE_SET_STATION_IP SetStationIp;
  EFI_PXE_BASE_CODE_SET_PACKETS SetPackets;
  EFI_PXE_BASE_CODE_MODE *Mode;
} EFI_PXE_BASE_CODE_PROTOCOL;

Parameters

Revision The revision of the EFI_PXE_BASE_CODE_PROTOCOL. All future revisions must be backwards compatible. If a future version is not backwards compatible it is not the same GUID.

Start Starts the PXE Base Code Protocol. Mode structure information is not valid and no other Base Code Protocol functions will operate until the Base Code is started. See the Start() function description.

Stop Stops the PXE Base Code Protocol. Mode structure information is unchanged by this function. No Base Code Protocol functions will operate until the Base Code is restarted. See the Stop() function description.

Dhcp Attempts to complete a DHCPv4 D.O.R.A. (discover / offer / request / acknowledge) or DHCPv6 S.A.R.R (solicit / advertise / request / reply) sequence. See the Dhcp() function description.

Discover Attempts to complete the PXE Boot Server and/or boot image discovery sequence. See the Discover() function description.

Mtftp Performs TFTP and MTFTP services. See the Mtftp() function description.

UdpWrite Writes a UDP packet to the network interface. See the UdpWrite() function description.

UdpRead Reads a UDP packet from the network interface. See the UdpRead() function description.

SetIpFilter Updates the IP receive filters of the network device. See the SetIpFilter() function description.

Arp Uses the ARP protocol to resolve a MAC address. See the Arp() function description.
SetParameters  Updates the parameters that affect the operation of the PXE Base Code Protocol. See the SetParameters() function description.

SetStationIp  Updates the station IP address and subnet mask values. See the SetStationIp() function description.

SetPackets  Updates the contents of the cached DHCP and Discover packets. See the SetPackets() function description.

Mode  Pointer to the EFI_PXE_BASE_CODE_MODE data for this device. The EFI_PXE_BASE_CODE_MODE structure is defined in “Related Definitions” below.

Related Definitions

//*******************************************************
// Maximum ARP and Route Entries
//*******************************************************
#define EFI_PXE_BASE_CODE_MAX_ARP_ENTRIES 8
#define EFI_PXE_BASE_CODE_MAX_ROUTE_ENTRIES 8

//*******************************************************
// EFI_PXE_BASE_CODE_MODE
//*******************************************************
typedef struct {
  BOOLEAN Started;
  BOOLEAN Ipv6Available;
  BOOLEAN Ipv6Supported;
  BOOLEAN UsingIpv6;
  BOOLEAN BisSupported;
  BOOLEAN BisDetected;
  BOOLEAN AutoArp;
  BOOLEAN SendGUID;
  BOOLEAN DhcpDiscoverValid;
  BOOLEAN DhcpAckReceived;
  BOOLEAN ProxyOfferReceived;
  BOOLEAN PxеDiscoverValid;
  BOOLEAN PxеReplyReceived;
  BOOLEAN PxеBisReplyReceived;
  BOOLEAN IcmpErrorReceived;
  BOOLEAN TftpErrorReceived;
  BOOLEAN MakeCallbacks;
  UINT8 TTL;
  UINT8 ToS;
  EFI_IP_ADDRESS StationIp;
  EFI_IP_ADDRESS SubnetMask;
  EFI_PXE_BASE_CODE_PACKET DhcpDiscover;
}
**EFI_PXE_BASE_CODE_PACKET**  
DhcpAck;

**EFI_PXE_BASE_CODE_PACKET**  
ProxyOffer;

**EFI_PXE_BASE_CODE_PACKET**  
PxeDiscover;

**EFI_PXE_BASE_CODE_PACKET**  
PxeReply;

**EFI_PXE_BASE_CODE_PACKET**  
PxeBisReply;

**EFI_PXE_BASE_CODE_IP_FILTER**  
IpFilter;

```c
UINT32 ArpCacheEntries;
```

```c
EFI_PXE_BASE_CODE_ARP_ENTRY
ArpCache[EFI_PXE_BASE_CODE_MAX_ARP_ENTRIES];
```

```c
UINT32 RouteTableEntries;
```

```c
EFI_PXE_BASE_CODE_ROUTE_ENTRY
RouteTable[EFI_PXE_BASE_CODE_MAX_ROUTE_ENTRIES];
```

```c
EFI_PXE_BASE_CODE_ICMP_ERROR IcmpError;
```

```c
EFI_PXE_BASE_CODE_TFTP_ERROR TftpError;
```

```c
} EFI_PXE_BASE_CODE_MODE;
```

**Started**  
`TRUE` if this device has been started by calling `Start()`. This field is set to `TRUE` by the `Start()` function and to `FALSE` by the `Stop()` function.

**Ipv6Available**  
`TRUE` if the UNDI protocol supports IPv6.

**Ipv6Supported**  
`TRUE` if this PXE Base Code Protocol implementation supports IPv6.

**UsingIpv6**  
`TRUE` if this device is currently using IPv6. This field is set by the `Start()` function.

**BisSupported**  
`TRUE` if this PXE Base Code implementation supports Boot Integrity Services (BIS). This field is set by the `Start()` function.

**BisDetected**  
`TRUE` if this device and the platform support Boot Integrity Services (BIS). This field is set by the `Start()` function.

**AutoArp**  
`TRUE` for automatic ARP packet generation; `FALSE` otherwise. This field is initialized to `TRUE` by `Start()` and can be modified with the `SetParameters()` function.

**SendGUID**  
This field is used to change the Client Hardware Address (chaddr) field in the DHCP and Discovery packets. Set to `TRUE` to send the SystemGuid (if one is available). Set to `FALSE` to send the client NIC MAC address. This field is initialized to `FALSE` by `Start()` and can be modified with the `SetParameters()` function.

**DhcpDiscoverValid**  
This field is initialized to `FALSE` by the `Start()` function and set to `TRUE` when the `Dhcp()` function completes successfully. When `TRUE`, the `DhcpDiscover` field is valid. This field can also be changed by the `SetPackets()` function.

**DhcpAckReceived**  
This field is initialized to `FALSE` by the `Start()` function and set to `TRUE` when the `Dhcp()` function completes successfully. When `TRUE`, the `DhcpAck` field is valid. This field can also be changed by the `SetPackets()` function.

**ProxyOfferReceived**  
This field is initialized to `FALSE` by the `Start()` function and set to `TRUE` when the `Dhcp()` function completes successfully and a proxy DHCP offer packet was received. When `TRUE`, the `ProxyOffer`
packet field is valid. This field can also be changed by the SetPackets() function.

**PxeDiscoverValid**

When TRUE, the PxeDiscover packet field is valid. This field is set to FALSE by the Start() and Dhcp() functions, and can be set to TRUE or FALSE by the Discover() and SetPackets() functions.

**PxeReplyReceived**

When TRUE, the PxeReply packet field is valid. This field is set to FALSE by the Start() and Dhcp() functions, and can be set to TRUE or FALSE by the Discover() and SetPackets() functions.

**PxeBisReplyReceived**

When TRUE, the PxeBisReply packet field is valid. This field is set to FALSE by the Start() and Dhcp() functions, and can be set to TRUE or FALSE by the Discover() and SetPackets() functions.

**IcmpErrorReceived**

Indicates whether the IcmpError field has been updated. This field is reset to FALSE by the Start(), Dhcp(), Discover(), Mtftp(), UdpRead(), UdpWrite(), and Arp() functions. If an ICMP error is received, this field will be set to TRUE after the IcmpError field is updated.

**TftpErrorReceived**

Indicates whether the TftpError field has been updated. This field is reset to FALSE by the Start() and Mtftp() functions. If a TFTP error is received, this field will be set to TRUE after the TftpError field is updated.

**MakeCallbacks**

When FALSE, callbacks will not be made. When TRUE, make callbacks to the PXE Base Code Callback Protocol. This field is reset to FALSE by the Start() function if the PXE Base Code Callback Protocol is not available. It is reset to TRUE by the Start() function if the PXE Base Code Callback Protocol is available.

**TTL**

The “time to live” field of the IP header. This field is initialized to DEFAULT_TTL (See “Related Definitions”) by the Start() function and can be modified by the SetParameters() function.

**ToS**

The type of service field of the IP header. This field is initialized to DEFAULT_ToS (See “Related Definitions”) by Start(), and can be modified with the SetParameters() function.

**StationIp**

The device’s current IP address. This field is initialized to a zero address by Start(). This field is set when the Dhcp() function completes successfully. This field can also be set by the SetStationIp() function. This field must be set to a valid IP address by either Dhcp() or SetStationIp() before the Discover(), Mtftp(), UdpRead(), UdpWrite(), and Arp() functions are called.

**SubnetMask**

The device’s current subnet mask. This field is initialized to a zero address by the Start() function. This field is set when the Dhcp() function completes successfully. This field can also be set by the SetStationIp() function. This field must be set to a valid subnet mask by either Dhcp() or SetStationIp() before the Discover(), Mtftp(), UdpRead(), UdpWrite(), or Arp() functions are called.

**DhcpDiscover**

Cached DHCP Discover packet. This field is zero-filled by the Start() function, and is set when the Dhcp() function completes
successfully. The contents of this field can replaced by the SetPackets() function.

**DhcpAck**  
Cached DHCP Ack packet. This field is zero-filled by the Start() function, and is set when the Dhcp() function completes successfully. The contents of this field can be replaced by the SetPackets() function.

**ProxyOffer**  
Cached Proxy Offer packet. This field is zero-filled by the Start() function, and is set when the Dhcp() function completes successfully. The contents of this field can be replaced by the SetPackets() function.

**PxeDiscover**  
Cached PXE Discover packet. This field is zero-filled by the Start() function, and is set when the Discover() function completes successfully. The contents of this field can be replaced by the SetPackets() function.

**PxeReply**  
Cached PXE Reply packet. This field is zero-filled by the Start() function, and is set when the Discover() function completes successfully. The contents of this field can be replaced by the SetPackets() function.

**PxeBisReply**  
Cached PXE BIS Reply packet. This field is zero-filled by the Start() function, and is set when the Discover() function completes successfully. This field can be replaced by the SetPackets() function.

**IpFilter**  
The current IP receive filter settings. The receive filter is disabled and the number of IP receive filters is set to zero by the Start() function, and is set by the SetIpFilter() function.

**ArpCacheEntries**  
The number of valid entries in the ARP cache. This field is reset to zero by the Start() function.

**ArpCache**  
Array of cached ARP entries.

**RouteTableEntries**  
The number of valid entries in the current route table. This field is reset to zero by the Start() function.

**RouteTable**  
Array of route table entries.

**IcmpError**  
ICMP error packet. This field is updated when an ICMP error is received and is undefined until the first ICMP error is received. This field is zero-filled by the Start() function.

**TftpError**  
TFTP error packet. This field is updated when a TFTP error is received and is undefined until the first TFTP error is received. This field is zero-filled by the Start() function.
//EFI_PXE_BASE_CODE_UDP_PORT
//******************************************************************************
typedef UINT16 EFI_PXE_BASE_CODE_UDP_PORT;

//EFI_IPv4_ADDRESS and EFI_IPv6_ADDRESS
//******************************************************************************
typedef struct {
    UINT8 Addr[4];
} EFI_IPv4_ADDRESS;

typedef struct {
    UINT8 Addr[16];
} EFI_IPv6_ADDRESS;

//EFI_IP_ADDRESS
//******************************************************************************
typedef union {
    UINT32 Addr[4];
    EFI_IPv4_ADDRESS v4;
    EFI_IPv6_ADDRESS v6;
} EFI_IP_ADDRESS;

//EFI_MAC_ADDRESS
//******************************************************************************
typedef struct {
    UINT8 Addr[32];
} EFI_MAC_ADDRESS;

**DHCP Packet Data Types**
This section defines the data types for DHCP packets, ICMP error packets, and TFTP error packets. All of these are byte-packed data structures.

**Note:** *All the multibyte fields in these structures are stored in network order.*
typedef struct {
    UINT8    BootpOpcode;
    UINT8    BootpHwType;
    UINT8    BootpHwAddrLen;
    UINT8    BootpGateHops;
    UINT32   BootpIdent;
    UINT16   BootpSeconds;
    UINT16   BootpFlags;
    UINT8    BootpCiAddr[4];
    UINT8    BootpYiAddr[4];
    UINT8    BootpSiAddr[4];
    UINT8    BootpGiAddr[4];
    UINT8    BootpHwAddr[16];
    UINT8    BootpSrvName[64];
    UINT8    BootpBootFile[128];
    UINT32   DhcpMagik;
    UINT8    DhcpOptions[56];
} EFI_PXE_BASE_CODE_DHCPV4_PACKET;

typedef struct {
    UINT32   MessageType:8;
    UINT32   TransactionId:24;
    UINT8    DhcpOptions[1024];
} EFI_PXE_BASE_CODE_DHCPV6_PACKET;

typedef union {
    UINT8    Raw[1472];
    EFI_PXE_BASE_CODE_DHCPV4_PACKET Dhcpv4;
    EFI_PXE_BASE_CODE_DHCPV6_PACKET Dhcpv6;
} EFI_PXE_BASE_CODE_PACKET;

typedef struct {
    UINT8    Type;
    UINT8    Code;
} EFI_PXE_BASE_CODE_ICMP_ERROR;
IP Receive Filter Settings
This section defines the data types for IP receive filter settings.

```c
#define EFI_PXE_BASE_CODE_MAX_IPCNT

//*******************************************************
// EFI_PXE_BASE_CODE_IP_FILTER
//*******************************************************
typedef struct {
    UINT8     Filters;
    UINT8     IpCnt;
    UINT16    reserved;
    EFI_IP_ADDRESS IpList[EFI_PXE_BASE_CODE_MAX_IPCNT];
} EFI_PXE_BASE_CODE_IP_FILTER;

#define EFI_PXE_BASE_CODE_IP_FILTER_STATION_IP            0x0001
#define EFI_PXE_BASE_CODE_IP_FILTER_BROADCAST             0x0002
#define EFI_PXE_BASE_CODE_IP_FILTER_PROMISCUOUS           0x0004
#define EFI_PXE_BASE_CODE_IP_FILTER_PROMISCUOUS_MULTICAST 0x0008
```

ARP Cache Entries
This section defines the data types for ARP cache entries, and route table entries.
typedef struct {
  EFI_IP_ADDRESS  IpAddr;
  EFI_MAC_ADDRESS MacAddr;
} EFI_PXE_BASE_CODE_ARP_ENTRY;

typedef struct {
  EFI_IP_ADDRESS  IpAddr;
  EFI_IP_ADDRESS  SubnetMask;
  EFI_IP_ADDRESS  GwAddr;
} EFI_PXE_BASE_CODE_ROUTE_ENTRY;

Filter Operations for UDP Read/Write Functions
This section defines the types of filter operations that can be used with the UdpRead() and UdpWrite() functions.

#define EFI_PXE_BASE_CODE_UDP_OPFLAGS_ANY_SRC_IP    0x0001
#define EFI_PXE_BASE_CODE_UDP_OPFLAGS_ANY_SRC_PORT  0x0002
#define EFI_PXE_BASE_CODE_UDP_OPFLAGS_ANY_DEST_IP   0x0004
#define EFI_PXE_BASE_CODE_UDP_OPFLAGS_ANY_DEST_PORT 0x0008
#define EFI_PXE_BASE_CODE_UDP_OPFLAGS_USE_FILTER    0x0010
#define EFI_PXE_BASE_CODE_UDP_OPFLAGS_MAY_FRAGMENT  0x0020
#define DEFAULT_TTL                                 16
#define DEFAULT_ToS                                 0

The following table defines values for the PXE DHCP and Bootserver Discover packet tags that are specific to the UEFI environment.  Complete definitions of all PXE tags are defined in Table 24-1 “PXE DHCP Options (Full List),” in the PXE Specification.

Table 24-1 PXE Tag Definitions for EFI

<table>
<thead>
<tr>
<th>Tag Name</th>
<th>Tag #</th>
<th>Length</th>
<th>Data Field</th>
</tr>
</thead>
<tbody>
<tr>
<td>Client Network Interface Identifier</td>
<td>94 [0x5E]</td>
<td>3 [0x03]</td>
<td>Type (1), MajorVer (1), MinorVer (1) Type is a one byte field that identifies the network interface that will be used by the downloaded program. Type is followed by two one byte version number fields, MajorVer and MinorVer. Type UNDI (1) = 0x01 Versions WIM-1.1a 16-bit UNDI: MajorVer = 0x02, MinorVer = 0x00 PXE-2.0 16-bit UNDI: MajorVer = 0x02, MinorVer = 0x01 32/64-bit UNDI &amp; H/W UNDI: MajorVer = 0x03, MinorVer = 0x00</td>
</tr>
</tbody>
</table>
Description

The basic mechanisms and flow for remote booting in UEFI are identical to the remote boot functionality described in detail in the PXE Specification. However, the actual execution environment, linkage, and calling conventions are replaced and enhanced for the UEFI environment.

The DHCP Option for the Client System Architecture is used to inform the DHCP server if the client is a UEFI environment in supported systems. The server may use this information to provide default images if it does not have a specific boot profile for the client.

The DHCP Option for Client Network Interface Identifier is used to inform the DHCP server of the client underlying network interface information. If the NII protocol is present, such information will be acquired by this protocol. Otherwise, Type = 0x01, MajorVer=0x03, MinorVer=0x00 will be the default value.

A handle that supports EFI_PXE_BASE_CODE_PROTOCOL is required to support EFI_LOAD_FILE_PROTOCOL. The EFI_LOAD_FILE_PROTOCOL function LoadFile() is used by the firmware to load files from devices that do not support file system type accesses. Specifically, the firmware’s boot manager invokes LoadFile() with BootPolicy being TRUE when attempting to boot from the device. The firmware then loads and transfers control to the downloaded PXE boot image. Once the remote image is successfully loaded, it may utilize the EFI_PXE_BASE_CODE_PROTOCOL interfaces, or even the EFI_SIMPLE_NETWORK_PROTOCOL interfaces, to continue the remote process.

EFI_PXE_BASE_CODE_PROTOCOL.Start()

Summary

Enables the use of the PXE Base Code Protocol functions.
Prototype

typedef
EFI_STATUS
(EIFIAPI *EFI_PXE_BASE_CODE_START) (  
  IN EFI_PXE_BASE_CODE_PROTOCOL *This,
  IN BOOLEAN UseIpv6
);

Parameters

This Pointer to the EFI_PXE_BASE_CODE_PROTOCOL instance.

UseIpv6 Specifies the type of IP addresses that are to be used during the session that is being started. Set to TRUE for IPv6 addresses, and FALSE for IPv4 addresses.

Description

This function enables the use of the PXE Base Code Protocol functions. If the Started field of the EFI_PXE_BASE_CODE_MODE structure is already TRUE, then EFI_ALREADY_STARTED will be returned. If UseIpv6 is TRUE, then IPv6 formatted addresses will be used in this session. If UseIpv6 is FALSE, then IPv4 formatted addresses will be used in this session. If UseIpv6 is TRUE, and theIpv6Supported field of the EFI_PXE_BASE_CODE_MODE structure is FALSE, then EFI_UNSUPPORTED will be returned. If there is not enough memory or other resources to start the PXE Base Code Protocol, then EFI_OUT_OF_RESOURCES will be returned. Otherwise, the PXE Base Code Protocol will be started, and all of the fields of the EFI_PXE_BASE_CODE_MODE structure will be initialized as follows:

- Started Set to TRUE.
- Ipv6Supported Unchanged.
- Ipv6Available Unchanged.
- UsingIpv6 Set to UseIpv6.
- BisSupported Unchanged.
- BisDetected Unchanged.
- AutoArp Set to TRUE.
- SendGUID Set to FALSE.
- TTL Set to DEFAULT_TTL.
- ToS Set to DEFAULT_ToS.
- DhcpCompleted Set to FALSE.
- ProxyOfferReceived Set to FALSE.
- StationIp Set to an address of all zeros.
- SubnetMask Set to a subnet mask of all zeros.
- DhcpDiscover Zero-filled.
- DhcpAck Zero-filled.
- ProxyOffer Zero-filled.
- PxeDiscoverValid Set to FALSE.
- PxeDiscover Zero-filled.
PxeReplyValid Set to FALSE.
PxeReply Zero-filled.
PxeBisReplyValid Set to FALSE.
PxeBisReply Zero-filled.
IpFilter Set the Filters field to 0 and the IpCnt field to 0.
ArpCacheEntries Set to 0.
ArpCache Zero-filled.
RouteTableEntries Set to 0.
RouteTable Zero-filled.
IcmpErrorReceived Set to FALSE.
IcmpError Zero-filled.
TftpErroReceived Set to FALSE.
TftpError Zero-filled.
MakeCallbacks Set to TRUE if the PXE Base Code Callback Protocol is available. Set to FALSE if the PXE Base Code Callback Protocol is not available.

Status Codes Returned

<table>
<thead>
<tr>
<th>Code</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>EFI_SUCCESS</td>
<td>The PXE Base Code Protocol was started.</td>
</tr>
<tr>
<td>EFI_INVALID_PARAMETER</td>
<td>The This parameter is NULL or does not point to a valid EFI_PXE_BASE_CODE_PROTOCOL structure.</td>
</tr>
<tr>
<td>EFI_UNSUPPORTED</td>
<td>UseIpv6 is TRUE, but the Ipv6Supported field of the EFI_PXE_BASE_CODE_MODE structure is FALSE.</td>
</tr>
<tr>
<td>EFI_ALREADY_STARTED</td>
<td>The PXE Base Code Protocol is already in the started state.</td>
</tr>
<tr>
<td>EFI_DEVICE_ERROR</td>
<td>The network device encountered an error during this operation.</td>
</tr>
<tr>
<td>EFI_OUT_OF_RESOURCES</td>
<td>Could not allocate enough memory or other resources to start the PXE Base Code Protocol.</td>
</tr>
</tbody>
</table>

EFI_PXE_BASE_CODE_PROTOCOL.Stop()

Summary
Disables the use of the PXE Base Code Protocol functions.

Prototype
typedef
   EFI_STATUS
   (EFIAPI *EFI_PXE_BASE_CODE_STOP)(
       IN EFI_PXE_BASE_CODE_PROTOCOL  *This
   );

Parameters
This Pointer to the EFI_PXE_BASE_CODE_PROTOCOL instance.
Description
This function stops all activity on the network device. All the resources allocated in `Start()` are released, the `Started` field of the `EFI_PXE_BASE_CODE_MODE` structure is set to `FALSE` and `EFI_SUCCESS` is returned. If the `Started` field of the `EFI_PXE_BASE_CODE_MODE` structure is already `FALSE`, then `EFI_NOT_STARTED` will be returned.

Status Codes Returned

<table>
<thead>
<tr>
<th>Code</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>EFI_SUCCESS</td>
<td>The PXE Base Code Protocol was stopped.</td>
</tr>
<tr>
<td>EFI_NOT_STARTED</td>
<td>The PXE Base Code Protocol is already in the stopped state.</td>
</tr>
<tr>
<td>EFI_INVALID_PARAMETER</td>
<td>The <code>This</code> parameter is <code>NULL</code> or does not point to a valid <code>EFI_PXE_BASE_CODE_PROTOCOL</code> structure.</td>
</tr>
<tr>
<td>EFI_DEVICE_ERROR</td>
<td>The network device encountered an error during this operation.</td>
</tr>
</tbody>
</table>

**EFI_PXE_BASE_CODE_PROTOCOL.Dhcp()**

Summary
Attempts to complete a DHCPv4 D.O.R.A. (discover / offer / request / acknowledge) or DHCPv6 S.A.R.R (solicit / advertise / request / reply) sequence.

Prototype
```c
typedef EFI_STATUS
    (EFIAPI *EFI_PXE_BASE_CODE_DHCP) ( IN EFI_PXE_BASE_CODE_PROTOCOL  *This,
                                            IN BOOLEAN SortOffers           );
```

Parameters
- **This** Pointer to the `EFI_PXE_BASE_CODE_PROTOCOL` instance.
- **SortOffers** `TRUE` if the offers received should be sorted. Set to `FALSE` to try the offers in the order that they are received.

Description
This function attempts to complete the DHCP sequence. If this sequence is completed, then `EFI_SUCCESS` is returned, and the `DhcpCompleted, ProxyOfferReceived, StationIp, SubnetMask, DhcpDiscover, DhcpAck,` and `ProxyOffer` fields of the `EFI_PXE_BASE_CODE_MODE` structure are filled in.

If `SortOffers` is `TRUE`, then the cached DHCP offer packets will be sorted before they are tried. If `SortOffers` is `FALSE`, then the cached DHCP offer packets will be tried in the order in which they are received. Please see the Preboot Execution Environment (PXE) Specification for additional details on the implementation of DHCP.

This function can take at least 31 seconds to timeout and return control to the caller. If the DHCP sequence does not complete, then `EFI_TIMEOUT` will be returned.
If the Callback Protocol does not return `EFI_PXE_BASE_CODE_CALLBACK_STATUS_CONTINUE`, then the DHCP sequence will be stopped and `EFI_ABORTED` will be returned.

### Status Codes Returned

<table>
<thead>
<tr>
<th>Code</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>EFI_SUCCESS</td>
<td>Valid DHCP has completed.</td>
</tr>
<tr>
<td>EFI_NOT_STARTED</td>
<td>The PXE Base Code Protocol is in the stopped state.</td>
</tr>
<tr>
<td>EFI_INVALID_PARAMETER</td>
<td>The <code>This</code> parameter is <code>NULL</code> or does not point to a valid <code>EFI_PXE_BASE_CODE_PROTOCOL</code> structure.</td>
</tr>
<tr>
<td>EFI_DEVICE_ERROR</td>
<td>The network device encountered an error during this operation.</td>
</tr>
<tr>
<td>EFI_OUT_OF_RESOURCES</td>
<td>Could not allocate enough memory to complete the DHCP Protocol.</td>
</tr>
<tr>
<td>EFI_ABORTED</td>
<td>The callback function aborted the DHCP Protocol.</td>
</tr>
<tr>
<td>EFI_TIMEOUT</td>
<td>The DHCP Protocol timed out.</td>
</tr>
<tr>
<td>EFI_ICMP_ERROR</td>
<td>An ICMP error packet was received during the DHCP session. The ICMP error packet has been cached in the <code>EFI_PXE_BASE_CODE_MODE.IconError</code> packet structure. Information about ICMP packet contents can be found in RFC 792.</td>
</tr>
<tr>
<td>EFI_NO_RESPONSE</td>
<td>Valid PXE offer was not received.</td>
</tr>
</tbody>
</table>

### `EFI_PXE_BASE_CODE_PROTOCOL.Discover()`

#### Summary

Attempts to complete the PXE Boot Server and/or boot image discovery sequence.

#### Prototype

```c
typedef EFI_STATUS
(EFIAPI *EFI_PXE_BASE_CODE_DISCOVER) (  
    IN EFI_PXE_BASE_CODE_PROTOCOL This,  
    IN UINT16 Type,  
    IN UINT16 *Layer,  
    IN BOOLEAN UseBis,  
    IN EFI_PXE_BASE_CODE_DISCOVER_INFO *Info OPTIONAL  
);  
```

#### Parameters

- **This**
  - Pointer to the `EFI_PXE_BASE_CODE_PROTOCOL` instance.
- **Type**
  - The type of bootstrap to perform. See “Related Definitions” below.
- **Layer**
  - Pointer to the boot server layer number to discover, which must be `PXE_BOOT_LAYER_INITIAL` when a new server type is being discovered. This is the only layer type that will perform multicast and broadcast discovery. All other layer types will only perform unicast discovery. If the boot server changes `Layer`, then the new `Layer` will be returned.
- **UseBis**
  - `TRUE` if Boot Integrity Services are to be used. `FALSE` otherwise.
Info

Pointer to a data structure that contains additional information on the type of discovery operation that is to be performed. If this field is NULL, then the contents of the cached DhcpAck and ProxyOffer packets will be used.

Related Definitions

//*******************************************************
// Bootstrap Types
//*******************************************************
#define EFI_PXE_BASE_CODE_BOOT_TYPE_BOOTSTRAP     0
#define EFI_PXE_BASE_CODE_BOOT_TYPE_MS_WINNT_RIS   1
#define EFI_PXE_BASE_CODE_BOOT_TYPE_INTEL_LCM     2
#define EFI_PXE_BASE_CODE_BOOT_TYPE_DOSUNDI      3
#define EFI_PXE_BASE_CODE_BOOT_TYPE_NEC_ESMPRO    4
#define EFI_PXE_BASE_CODE_BOOT_TYPE_IBMWSoD       5
#define EFI_PXE_BASE_CODE_BOOT_TYPE_IBMLCCM       6
#define EFI_PXE_BASE_CODE_BOOT_TYPE_CA_UNICENTER_TNG 7
#define EFI_PXE_BASE_CODE_BOOT_TYPE_HP_OPENVIEW   8
#define EFI_PXE_BASE_CODE_BOOT_TYPE_ALTIRIS_9     9
#define EFI_PXE_BASE_CODE_BOOT_TYPE_ALTIRIS_10    10
#define EFI_PXE_BASE_CODE_BOOT_TYPE_ALTIRIS_11    11
#define EFI_PXE_BASE_CODE_BOOT_TYPE_NOT_USED_12   12
#define EFI_PXE_BASE_CODE_BOOT_TYPE_REDMAN_INSTALL 13
#define EFI_PXE_BASE_CODE_BOOT_TYPE_REDMAN_BOOT   14
#define EFI_PXE_BASE_CODE_BOOT_TYPE_REDMAN_REMBO   15
#define EFI_PXE_BASE_CODE_BOOT_TYPE_BEOB0OT       16
//
// Values 17 through 32767 are reserved.
// Values 32768 through 65279 are for vendor use.
// Values 65280 through 65534 are reserved.
//
#define EFI_PXE_BASE_CODE_BOOT_TYPE_PXETEST 65535

#define EFI_PXE_BASE_CODE_BOOT_LAYER_MASK    0x7FFF
#define EFI_PXE_BASE_CODE_BOOT_LAYER_INITIAL 0x0000

//*******************************************************
// EFI_PXE_BASE_CODE_DISCOVER_INFO
//*******************************************************
typedef struct {
    BOOLEAN UseMCast;
    BOOLEAN UseBCast;
    BOOLEAN UseUCast;
    BOOLEAN MustUseList;
    EFI_IP_ADDRESS ServerMCastIp;
    UINT16 IpCnt;
} DiscoverInfo;
typedef struct {
    UINT16 Type;
    BOOLEAN AcceptAnyResponse;
    UINT8 reserved;
    EFI_IP_ADDRESS IpAddr;
} EFI_PXE_BASE_CODE_SRVLIST;

Description

This function attempts to complete the PXE Boot Server and/or boot image discovery sequence. If this sequence is completed, then EFI_SUCCESS is returned, and the PxeDiscoverValid, PxeDiscover, PxeReplyReceived, and PxeReply fields of the EFI_PXE_BASE_CODE_MODE structure are filled in. If UseBis is TRUE, then the PxeBisReplyReceived and PxeBisReply fields of the EFI_PXE_BASE_CODE_MODE structure will also be filled in. If UseBis is FALSE, then PxeBisReplyValid will be set to FALSE.

In the structure referenced by parameter Info, the PXE Boot Server list, SrvList[], has two uses: It is the Boot Server IP address list used for unicast discovery (if the UseUCast field is TRUE), and it is the list used for Boot Server verification (if the MustUseList field is TRUE). Also, if the MustUseList field in that structure is TRUE and the AcceptAnyResponse field in the SrvList[] array is TRUE, any Boot Server reply of that type will be accepted. If the AcceptAnyResponse field is FALSE, only responses from Boot Servers with matching IP addresses will be accepted.

This function can take at least 10 seconds to timeout and return control to the caller. If the Discovery sequence does not complete, then EFI_TIMEOUT will be returned. Please see the Preboot Execution Environment (PXE) Specification for additional details on the implementation of the Discovery sequence.

If the Callback Protocol does not return EFI_PXE_BASE_CODE_CALLBACK_STATUS_CONTINUE, then the Discovery sequence is stopped and EFI_ABORTED will be returned.
## Status Codes Returned

<table>
<thead>
<tr>
<th>Code</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>EFI_SUCCESS</td>
<td>The Discovery sequence has been completed.</td>
</tr>
<tr>
<td>EFI_NOT_STARTED</td>
<td>The PXE Base Code Protocol is in the stopped state.</td>
</tr>
<tr>
<td>EFI_INVALID_PARAMETER</td>
<td>One or more of the following conditions was <strong>TRUE</strong>:</td>
</tr>
<tr>
<td></td>
<td>- The <code>This</code> parameter was <strong>NULL</strong></td>
</tr>
<tr>
<td></td>
<td>- The <code>This</code> parameter did not point to a valid <code>EFI_PXE_BASE_CODE_PROTOCOL</code> structure</td>
</tr>
<tr>
<td></td>
<td>- The <code>Layer</code> parameter was <strong>NULL</strong></td>
</tr>
<tr>
<td></td>
<td>- The <code>Info-&gt;ServerMCastIp</code> parameter does not contain a valid multicast IP address</td>
</tr>
<tr>
<td></td>
<td>- The <code>Info-&gt;UseUCast</code> parameter is not <strong>FALSE</strong> and the <code>Info-&gt;IpCnt</code> parameter is zero</td>
</tr>
<tr>
<td></td>
<td>One or more of the IP addresses in the <code>Info-&gt;SrvList[]</code> array is not a valid unicast IP address.</td>
</tr>
<tr>
<td>EFIDEVICE_ERROR</td>
<td>The network device encountered an error during this operation.</td>
</tr>
<tr>
<td>EFI_OUT_OF_RESOURCES</td>
<td>Could not allocate enough memory to complete Discovery.</td>
</tr>
<tr>
<td>EFI_ABORTED</td>
<td>The callback function aborted the Discovery sequence.</td>
</tr>
<tr>
<td>EFI_TIMEOUT</td>
<td>The Discovery sequence timed out.</td>
</tr>
<tr>
<td>EFI_ICMP_ERROR</td>
<td>An ICMP error packet was received during the PXE discovery session. The ICMP error packet has been cached in the <code>EFI_PXE_BASE_CODE_MODE.IcmpError</code> packet structure. Information about ICMP packet contents can be found in RFC 792.</td>
</tr>
</tbody>
</table>

### EFI_PXE_BASE_CODE_PROTOCOL.Mtftp() Summary

Used to perform TFTP and MTFTP services.

### Prototype

```c
typedef
  EFI_STATUS
  (EFIAPI *EFI_PXE_BASE_CODE_MTFTP) (
    IN EFI_PXE_BASE_CODE_PROTOCOL *This,
    IN EFI_PXE_BASE_CODE_TFTP_OPCODE Operation,
    IN OUT VOID *BufferPtr, OPTIONAL
    IN BOOLEAN Overwrite,
    IN OUT UINT64 *BufferSize,
    IN UINTN *BlockSize, OPTIONAL
    IN EFI_IP_ADDRESS *ServerIp,
    IN CHAR8 *Filename, OPTIONAL
    IN EFI_PXE_BASE_CODE_MTFTP_INFO *Info, OPTIONAL
    IN BOOLEAN DontUseBuffer
  );
```
Parameters

This
- Pointer to the **EFI_PXE_BASE_CODE_PROTOCOL** instance.

Operation
- The type of operation to perform. See “Related Definitions” below for the list of operation types.

BufferPtr
- A pointer to the data buffer. Ignored for read file if **DontUseBuffer** is **TRUE**.

Overwrite
- Only used on write file operations. **TRUE** if a file on a remote server can be overwritten.

BufferSize
- For get-file-size operations, **BufferSize** returns the size of the requested file. For read-file and write-file operations, this parameter is set to the size of the buffer specified by the **BufferPtr** parameter. For read-file operations, if **EFI_BUFFER_TOO_SMALL** is returned, **BufferSize** returns the size of the requested file.

BlockSize
- The requested block size to be used during a TFTP transfer. This must be at least 512. If this field is **NULL**, then the largest block size supported by the implementation will be used.

ServerIp
- The TFTP / MTFTP server IP address.

Filename
- A Null-terminated ASCII string that specifies a directory name or a file name. This is ignored by MTFTP read directory.

Info
- Pointer to the MTFTP information. This information is required to start or join a multicast TFTP session. It is also required to perform the “get file size” and “read directory” operations of MTFTP. See “Related Definitions” below for the description of this data structure.

DontUseBuffer
- Set to **FALSE** for normal TFTP and MTFTP read file operation. Setting this to **TRUE** will cause TFTP and MTFTP read file operations to function without a receive buffer, and all of the received packets are passed to the Callback Protocol which is responsible for storing them. This field is only used by TFTP and MTFTP read file.
Related Definitions

```c
//***********************************************************************
// EFI_PXE_BASE_CODE_TFTP_OPCODE
//***********************************************************************
typedef enum {
  EFI_PXE_BASE_CODE_TFTP_FIRST,
  EFI_PXE_BASE_CODE_TFTP_GET_FILE_SIZE,
  EFI_PXE_BASE_CODE_TFTP_READ_FILE,
  EFI_PXE_BASE_CODE_TFTP_WRITE_FILE,
  EFI_PXE_BASE_CODE_TFTP_READ_DIRECTORY,
  EFI_PXE_BASE_CODE_MTFTP_GET_FILE_SIZE,
  EFI_PXE_BASE_CODE_MTFTP_READ_FILE,
  EFI_PXE_BASE_CODE_MTFTP_READ_DIRECTORY,
  EFI_PXE_BASE_CODE_MTFTP_LAST
} EFI_PXE_BASE_CODE_TFTP_OPCODE;
```

```c
//***********************************************************************
// EFI_PXE_BASE_CODE_MTFTP_INFO
//***********************************************************************
typedef struct {
  EFI_IP_ADDRESS MCastIp;
  EFI_PXE_BASE_CODE_UDP_PORT CPort;
  EFI_PXE_BASE_CODE_UDP_PORT SPort;
  UINT16 ListenTimeout;
  UINT16 TransmitTimeout;
} EFI_PXE_BASE_CODE_MTFTP_INFO;
```

**MCastIp**  
File multicast IP address. This is the IP address to which the server will send the requested file.

**CPort**  
Client multicast listening port. This is the UDP port to which the server will send the requested file.

**SPort**  
Server multicast listening port. This is the UDP port on which the server listens for multicast open requests and data acks.

**ListenTimeout**  
The number of seconds a client should listen for an active multicast session before requesting a new multicast session.

**TransmitTimeout**  
The number of seconds a client should wait for a packet from the server before retransmitting the previous open request or data ack packet.

**Description**

This function is used to perform TFTP and MTFTP services. This includes the TFTP operations to get the size of a file, read a directory, read a file, and write a file. It also includes the MTFTP operations to get the size of a file, read a directory, and read a file. The type of operation is specified by **Operation**. If the callback function that is invoked during the TFTP/MTFTP operation does not return **EFI_PXE_BASE_CODE_CALLBACK_STATUS_CONTINUE**, then **EFI_ABORTED** will be returned.
For read operations, the return data will be placed in the buffer specified by `BufferPtr`. If `BufferSize` is too small to contain the entire downloaded file, then `EFI_BUFFER_TOO_SMALL` will be returned and `BufferSize` will be set to zero or the size of the requested file (the size of the requested file is only returned if the TFTP server supports TFTP options). If `BufferSize` is large enough for the read operation, then `BufferSize` will be set to the size of the downloaded file, and `EFI_SUCCESS` will be returned. Applications using the `PxeBc.Mtftp()` services should use the get-file-size operations to determine the size of the downloaded file prior to using the read-file operations—especially when downloading large (greater than 64 MiB) files—instead of making two calls to the read-file operation. Following this recommendation will save time if the file is larger than expected and the TFTP server does not support TFTP option extensions. Without TFTP option extension support, the client has to download the entire file, counting and discarding the received packets, to determine the file size.

For write operations, the data to be sent is in the buffer specified by `BufferPtr`. `BufferSize` specifies the number of bytes to send. If the write operation completes successfully, then `EFI_SUCCESS` will be returned.

For TFTP “get file size” operations, the size of the requested file or directory is returned in `BufferSize`, and `EFI_SUCCESS` will be returned. If the TFTP server does not support options, the file will be downloaded into a bit bucket and the length of the downloaded file will be returned. For MTFTP “get file size” operations, if the MTFTP server does not support the “get file size” option, `EFI_UNSUPPORTED` will be returned.

This function can take up to 10 seconds to timeout and return control to the caller. If the TFTP sequence does not complete, `EFI_TIMEOUT` will be returned.

If the Callback Protocol does not return `EFI_PXE_BASE_CODE_CALLBACK_STATUS_CONTINUE`, then the TFTP sequence is stopped and `EFI_ABORTED` will be returned.

The format of the data returned from a TFTP read directory operation is a null-terminated filename followed by a null-terminated information string, of the form “size year-month-day hour:minute:second” (i.e., %d %d-%d-%d %d:%d:%f - note that the seconds field can be a decimal number), where the date and time are UTC. For an MTFTP read directory command, there is additionally a null-terminated multicast IP address preceding the filename of the form %d.%d.%d.%d for IP v4. The final entry is itself null-terminated, so that the final information string is terminated with two null octets.
## Status Codes Returned

<table>
<thead>
<tr>
<th>Code Description</th>
<th>Explanation</th>
</tr>
</thead>
<tbody>
<tr>
<td>EFI_SUCCESS</td>
<td>The TFTP/MTFTP operation was completed.</td>
</tr>
<tr>
<td>EFI_NOT_STARTED</td>
<td>The PXE Base Code Protocol is in the stopped state.</td>
</tr>
</tbody>
</table>
| EFI_INVALID_PARAMETER | One or more of the following conditions was **TRUE:**  
- The `This` parameter was **NULL**  
- The `This` parameter did not point to a valid `EFI_PXE_BASE_CODE_PROTOCOL` structure  
- The Operation parameter was not one of the listed `EFI_PXE_BASE_CODE_TFTP_OPCODE` constants  
- The `BufferPtr` parameter was **NULL** and the `DontUseBuffer` parameter was **FALSE**  
- The `BufferSize` parameter was **NULL**  
- The `BlockSize` parameter was not **NULL** and `*BlockSize` was less than 512  
- The ServerIp parameter was **NULL** or did not contain a valid unicast IP address  
- The Filename parameter was **NULL** for a file transfer or information request  
- The `Info` parameter was **NULL** for a multicast request  
- The `Info->MCastIp` parameter is not a valid multicast IP address |
| EFI_DEVICE_ERROR | The network device encountered an error during this operation. |
| EFI_BUFFER_TOO_SMALL | The buffer is not large enough to complete the read operation. |
| EFI_ABORTED      | The callback function aborted the TFTP/MTFTP operation. |
| EFI_TIMEOUT      | The TFTP/MTFTP operation timed out. |
| EFI_TFTP_ERROR   | A TFTP error packet was received during the MTFTP session. The TFTP error packet has been cached in the `EFI_PXE_BASE_CODE_MODE.TftpError` packet structure. Information about TFTP error packet contents can be found in RFC 1350. |
| EFI_ICMP_ERROR   | An ICMP error packet was received during the MTFTP session. The ICMP error packet has been cached in the `EFI_PXE_BASE_CODE_MODE.IcmpError` packet structure. Information about ICMP packet contents can be found in RFC 792. |

### EFI_PXE_BASE_CODE_PROTOCOL.UdpWrite()

**Summary**

 Writes a UDP packet to the network interface.
Prototype

typedef
EFI_STATUS
(EIFIAPIC*EFI_PXE_BASE_CODE_UDP_WRITE)
(
    IN EFI_PXE_BASE_CODE_PROTOCOL *This,
    IN UINT16 OpFlags,
    IN EFI_IP_ADDRESS *DestIp,
    IN EFI_PXE_BASE_CODE_UDP_PORT *DestPort,
    IN EFI_IP_ADDRESS *GatewayIp,
    OPTIONAL IN EFI_IP_ADDRESS *SrcIp,
    OPTIONAL IN OUT EFI_PXE_BASE_CODE_UDP_PORT *SrcPort,
    OPTIONAL IN UINTN *HeaderSize,
    OPTIONAL IN VOID *HeaderPtr,
    OPTIONAL IN UINTN *BufferSize,
    IN VOID *BufferPtr
);

Parameters

This Pointer to the EFI_PXE_BASE_CODE_PROTOCOL instance.
OpFlags The UDP operation flags. If MAY_FRAGMENT is set, then if required, this UDP write operation may be broken up across multiple packets.
DestIp The destination IP address.
DestPort The destination UDP port number.
GatewayIp The gateway IP address. If DestIp is not in the same subnet as StationIp, then this gateway IP address will be used. If this field is NULL, and the DestIp is not in the same subnet as StationIp, then the RouteTable will be used.
SrcIp The source IP address. If this field is NULL, then StationIp will be used as the source IP address.
SrcPort The source UDP port number. If OpFlags has ANY_SRC_PORT set or SrcPort is NULL, then a source UDP port will be automatically selected. If a source UDP port was automatically selected, and SrcPort is not NULL, then it will be returned in SrcPort.
HeaderSize An optional field which may be set to the length of a header at HeaderPtr to be prefixed to the data at BufferPtr.
HeaderPtr If HeaderSize is not NULL, a pointer to a header to be prefixed to the data at BufferPtr.
BufferSize A pointer to the size of the data at BufferPtr.
BufferPtr A pointer to the data to be written.

Description

This function writes a UDP packet specified by the (optional HeaderPtr and) BufferPtr parameters to the network interface. The UDP header is automatically built by this routine. It uses the parameters OpFlags, DestIp, DestPort, GatewayIp, SrcIp, and SrcPort to build this header. If the packet is successfully built and transmitted through the network interface, then EFI_SUCCESS will be returned. If a
timeout occurs during the transmission of the packet, then \texttt{EFI_TIMEOUT} will be returned. If an ICMP
error occurs during the transmission of the packet, then the \texttt{IcmpErrorReceived} field is set to \texttt{TRUE},
the \texttt{IcmpError} field is filled in and \texttt{EFI_ICMP_ERROR} will be returned. If the Callback Protocol does not
return \texttt{EFI_PXE_BASE_CODE_CALLBACK_STATUS_CONTINUE}, then \texttt{EFI_ABORTED} will be returned.

### Status Codes Returned

<table>
<thead>
<tr>
<th>Status Code</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>\texttt{EFI_SUCCESS}</td>
<td>The UDP Write operation was completed.</td>
</tr>
<tr>
<td>\texttt{EFI_NOT_STARTED}</td>
<td>The PXE Base Code Protocol is in the stopped state.</td>
</tr>
<tr>
<td>\texttt{EFI_INVALID_PARAMETER}</td>
<td>One or more of the following conditions was \texttt{TRUE}:</td>
</tr>
<tr>
<td></td>
<td>• The \texttt{This} parameter was \texttt{NULL}</td>
</tr>
<tr>
<td></td>
<td>• The \texttt{This} parameter did not point to a valid \texttt{EFI_PXE_BASE_CODE_PROTOCOL} structure</td>
</tr>
<tr>
<td></td>
<td>• Reserved bits in the OpFlags parameter were not set to zero</td>
</tr>
<tr>
<td></td>
<td>• The DestIp parameter was \texttt{NULL}</td>
</tr>
<tr>
<td></td>
<td>• The DestPort parameter was \texttt{NULL}</td>
</tr>
<tr>
<td></td>
<td>• The GatewayIp parameter was not \texttt{NULL} and did not contain a valid unicast IP address.</td>
</tr>
<tr>
<td></td>
<td>• The HeaderSize parameter was not \texttt{NULL} and *HeaderSize is zero</td>
</tr>
<tr>
<td></td>
<td>• The *HeaderSize parameter was not zero and the HeaderPtr parameter was \texttt{NULL}</td>
</tr>
<tr>
<td></td>
<td>• The BufferSize parameter was \texttt{NULL}</td>
</tr>
<tr>
<td></td>
<td>• The *BufferSize parameter was not zero and the BufferPtr parameter was \texttt{NULL}</td>
</tr>
<tr>
<td>\texttt{EFI_DEVICE_ERROR}</td>
<td>The network device encountered an error during this operation.</td>
</tr>
<tr>
<td>\texttt{EFI_BAD_BUFFER_SIZE}</td>
<td>The buffer is too long to be transmitted.</td>
</tr>
<tr>
<td>\texttt{EFI_ABORTED}</td>
<td>The callback function aborted the UDP Write operation.</td>
</tr>
<tr>
<td>\texttt{EFI_TIMEOUT}</td>
<td>The UDP Write operation timed out.</td>
</tr>
<tr>
<td>\texttt{EFI_ICMP_ERROR}</td>
<td>An ICMP error packet was received during the UDP write session. The ICMP error packet has been cached in the \texttt{EFI_PXE_BASE_CODE_MODE.IcmpError} packet structure. Information about ICMP packet contents can be found in RFC 792.</td>
</tr>
</tbody>
</table>

**

\texttt{EFI_PXE_BASE_CODE_PROTOCOL.UdpRead()}**

**Summary**

Reads a UDP packet from the network interface.
Prototype

```c
typedef
EFI_STATUS
(EFIAPI *EFI_PXE_BASE_CODE_UDP_READ) (  
    IN EFI_PXE_BASE_CODE_PROTOCOL *This,  
    IN UINT16 OpFlags,  
    IN OUT EFI_IP_ADDRESS *DestIp,  
    IN OUT EFI_IP_ADDRESS *SrcIp,  
    OPTIONAL IN OUT EFI_PXE_BASE_CODE_UDP_PORT *DestPort,  
    OPTIONAL IN OUT EFI_PXE_BASE_CODE_UDP_PORT *SrcPort,  
    OPTIONAL IN UINTN *HeaderSize,  
    OPTIONAL IN VOID *HeaderPtr,  
    OPTIONAL IN VOID *BufferPtr,  
    IN OUT UINTN *BufferSize,  
    IN VOID *BufferPtr  
);
```

Parameters

- **This**: Pointer to the `EFI_PXE_BASE_CODE_PROTOCOL` instance.
- **OpFlags**: The UDP operation flags.
- **DestIp**: The destination IP address.
- **DestPort**: The destination UDP port number.
- **SrcIp**: The source IP address.
- **SrcPort**: The source UDP port number.
- **HeaderSize**: An optional field which may be set to the length of a header to be put in `HeaderPtr`.
- **HeaderPtr**: If `HeaderSize` is not `NULL`, a pointer to a buffer to hold the `HeaderSize` bytes which follow the UDP header.
- **BufferSize**: On input, a pointer to the size of the buffer at `BufferPtr`. On output, the size of the data written to `BufferPtr`.
- **BufferPtr**: A pointer to the data to be read.

Description

This function reads a UDP packet from a network interface. The data contents are returned in (the optional `HeaderPtr` and) `BufferPtr`, and the size of the buffer received is returned in `BufferSize`. If the input `BufferSize` is smaller than the UDP packet received (less optional `HeaderSize`), it will be set to the required size, and `EFI_BUFFER_TOO_SMALL` will be returned. In this case, the contents of `BufferPtr` are undefined, and the packet is lost. If a UDP packet is successfully received, then `EFI_SUCCESS` will be returned, and the information from the UDP header will be returned in `DestIp`, `DestPort`, `SrcIp`, and `SrcPort` if they are not `NULL`. Depending on the values of `OpFlags` and the `DestIp`, `DestPort`, `SrcIp`, and `SrcPort` input values, different types of UDP packet receive filtering will be performed. The following tables summarize these receive filter operations.
### Table 24-2 Destination IP Filter Operation

<table>
<thead>
<tr>
<th>OpFlags</th>
<th>OpFlags</th>
<th>DestIp</th>
<th>Action</th>
</tr>
</thead>
<tbody>
<tr>
<td>USE_FILTER</td>
<td>ANY_DEST_IP</td>
<td>NULL</td>
<td>Receive a packet sent to <strong>StationIp</strong>.</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>NULL</td>
<td>Receive a packet sent to any IP address.</td>
</tr>
<tr>
<td>1</td>
<td>x</td>
<td>NULL</td>
<td>Receive a packet whose destination IP address passes the IP filter.</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>not NULL</td>
<td>Receive a packet whose destination IP address matches <strong>DestIp</strong>.</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>not NULL</td>
<td>Receive a packet sent to any IP address and, return the destination IP address in <strong>DestIp</strong>.</td>
</tr>
<tr>
<td>1</td>
<td>x</td>
<td>not NULL</td>
<td>Receive a packet whose destination IP address passes the IP filter, and return the destination IP address in <strong>DestIp</strong>.</td>
</tr>
</tbody>
</table>

### Table 24-3 Destination UDP Port Filter Operation

<table>
<thead>
<tr>
<th>OpFlags</th>
<th>DestPort</th>
<th>Action</th>
</tr>
</thead>
<tbody>
<tr>
<td>ANY_DEST_PORT</td>
<td>NULL</td>
<td>Return <strong>EFI_INVALID_PARAMETER</strong></td>
</tr>
<tr>
<td>0</td>
<td>NULL</td>
<td>Receive a packet sent to any UDP port.</td>
</tr>
<tr>
<td>0</td>
<td>not NULL</td>
<td>Receive a packet whose destination Port matches <strong>DestPort</strong>.</td>
</tr>
<tr>
<td>1</td>
<td>not NULL</td>
<td>Receive a packet sent to any UDP port, and return the destination port in <strong>DestPort</strong>.</td>
</tr>
</tbody>
</table>

### Table 24-4 Source IP Filter Operation

<table>
<thead>
<tr>
<th>OpFlags</th>
<th>SrcIp</th>
<th>Action</th>
</tr>
</thead>
<tbody>
<tr>
<td>ANY_SRC_IP</td>
<td>NULL</td>
<td>Return <strong>EFI_INVALID_PARAMETER</strong></td>
</tr>
<tr>
<td>0</td>
<td>NULL</td>
<td>Receive a packet sent from any IP address.</td>
</tr>
<tr>
<td>0</td>
<td>not NULL</td>
<td>Receive a packet whose source IP address matches <strong>SrcIp</strong>.</td>
</tr>
<tr>
<td>1</td>
<td>not NULL</td>
<td>Receive a packet sent from any IP address, and return the source IP address in <strong>SrcIp</strong>.</td>
</tr>
</tbody>
</table>
Table 24-5 Source UDP Port Filter Operation

<table>
<thead>
<tr>
<th>OpFlags ANY_SRC_PORT</th>
<th>SrcPort</th>
<th>Action</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>NULL</td>
<td>Return <strong>EFI_INVALID_PARAMETER</strong></td>
</tr>
<tr>
<td>1</td>
<td>NULL</td>
<td>Receive a packet sent from any UDP port.</td>
</tr>
<tr>
<td>0</td>
<td>not NULL</td>
<td>Receive a packet whose source UDP port matches <strong>SrcPort</strong>.</td>
</tr>
<tr>
<td>1</td>
<td>not NULL</td>
<td>Receive a packet sent from any UDP port, and return the source UD port in <strong>SrcPort</strong>.</td>
</tr>
</tbody>
</table>

### Status Codes Returned

<table>
<thead>
<tr>
<th>Code</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>EFI_SUCCESS</strong></td>
<td>The UDP Read operation was completed.</td>
</tr>
<tr>
<td><strong>EFI_NOT_STARTED</strong></td>
<td>The PXE Base Code Protocol is in the stopped state.</td>
</tr>
<tr>
<td><strong>EFI_INVALID_PARAMETER</strong></td>
<td>One or more of the following conditions was <strong>TRUE</strong>:</td>
</tr>
<tr>
<td></td>
<td>• The <strong>This</strong> parameter was <strong>NULL</strong></td>
</tr>
<tr>
<td></td>
<td>• The <strong>This</strong> parameter did not point to a valid <strong>EFI_PXE_BASE_CODE_PROTOCOL</strong> structure</td>
</tr>
<tr>
<td></td>
<td>• Reserved bits in the OpFlags parameter were not set to zero</td>
</tr>
<tr>
<td></td>
<td>• The HeaderSize parameter is not <strong>NULL</strong> and *HeaderSize is zero</td>
</tr>
<tr>
<td></td>
<td>• The HeaderSize parameter is not <strong>NULL</strong>. <strong>HeaderPtr parameter is NULL</strong></td>
</tr>
<tr>
<td></td>
<td>• The BufferSize parameter is <strong>NULL</strong></td>
</tr>
<tr>
<td></td>
<td>• The BufferPtr parameter is <strong>NULL</strong></td>
</tr>
<tr>
<td><strong>EFI_DEVICE_ERROR</strong></td>
<td>The network device encountered an error during this operation.</td>
</tr>
<tr>
<td><strong>EFI_BUFFER_TOO_SMALL</strong></td>
<td>The packet is larger than <strong>Buffer</strong> can hold.</td>
</tr>
<tr>
<td><strong>EFI_ABORTED</strong></td>
<td>The callback function aborted the UDP Read operation.</td>
</tr>
<tr>
<td><strong>EFI_TIMEOUT</strong></td>
<td>The UDP Read operation timed out.</td>
</tr>
</tbody>
</table>

### EFI_PXE_BASE_CODE_PROTOCOL.SetIpFilter()

**Summary**

Updates the IP receive filters of a network device and enables software filtering.

**Prototype**

```c
typedef EFI_STATUS (EFIAPI *EFI_PXE_BASE_CODE_SET_IP_FILTER) (    
    IN EFI_PXE_BASE_CODE_PROTOCOL *This,                        
    IN EFI_PXE_BASE_CODE_IP_FILTER *NewFilter                  
);
```

**Parameters**

- **This**: Pointer to the **EFI_PXE_BASE_CODE_PROTOCOL** instance.
NewFilter

Pointer to the new set of IP receive filters.

Description
The **NewFilter** field is used to modify the network device’s current IP receive filter settings and to enable a software filter. This function updates the **IpFilter** field of the **EFI_PXE_BASE_CODE_MODE** structure with the contents of **NewIpFilter**. The software filter is used when the **USE_FILTER** in **OpFlags** is set to **UdpRead()**. The current hardware filter remains in effect no matter what the settings of **OpFlags** are, so that the meaning of **ANY_DEST_IP** set in **OpFlags** to **UdpRead()** is from those packets whose reception is enabled in hardware – physical NIC address (unicast), broadcast address, logical address or addresses (multicast), or all (promiscuous). **UdpRead()** does not modify the IP filter settings. **Dhcp()**, **Discover()**, and **Mtftp()** set the IP filter, and return with the IP receive filter list emptied and the filter set to **EFI_PXE_BASE_CODE_IP_FILTER_STATION_IP**. If an application or driver wishes to preserve the IP receive filter settings before these calls, and use **SetIpFilter()** to restore them after the calls. If incompatible filtering is requested (for example, **PROMISCUOUS** with anything else) or if the device does not support a requested filter setting and it cannot be accommodated in software (for example, **PROMISCUOUS** not supported), **EFI_INVALID_PARAMETER** will be returned. The **IPlist** field is used to enable IPs other than the **StationIP**. They may be multicast or unicast. If **IPcnt** is set as well as **EFI_PXE_BASE_CODE_IP_FILTER_STATION_IP**, then both the **StationIP** and the IPs from the **IPlist** will be used.

Status Codes Returned

<table>
<thead>
<tr>
<th>Code</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>EFI_SUCCESS</strong></td>
<td>The IP receive filter settings were updated.</td>
</tr>
</tbody>
</table>
| **EFI_INVALID_PARAMETER** | - One or more of the following conditions was **TRUE**:  
    - The **This** parameter was **NULL**  
    - The **This** parameter did not point to a valid **EFI_PXE_BASE_CODE_PROTOCOL** structure  
    - The **NewFilter** parameter was **NULL**  
    - The **NewFilter**-> **IPlist** [] array contains one or more broadcast IP addresses |
| **EFI_NOT_STARTED** | The PXE Base Code Protocol is not in the started state. |

**EFI_PXE_BASE_CODE_PROTOCOL.Arp()**

Summary
Uses the ARP protocol to resolve a MAC address.
Prototype

```c
Prototype

typedef
EFI_STATUS
(EIFIAPI *EFI_PXE_BASE_CODE_ARP) (
    IN EFI_PXE_BASE_CODE_PROTOCOL *This,
    IN EFI_IP_ADDRESS *IpAddr,
    IN EFI_MAC_ADDRESS *MacAddr
);
```

Parameters

- **This**: Pointer to the EFI_PXE_BASE_CODE_PROTOCOL instance.
- **IpAddr**: Pointer to the IP address that is used to resolve a MAC address. When the MAC address is resolved, the ArpCacheEntries and ArpCache fields of the EFI_PXE_BASE_CODE_MODE structure are updated.
- **MacAddr**: If not NULL, a pointer to the MAC address that was resolved with the ARP protocol.

Description

This function uses the ARP protocol to resolve a MAC address. The UsingIpv6 field of the EFI_PXE_BASE_CODE_MODE structure is used to determine if IPv4 or IPv6 addresses are being used. The IP address specified by IpAddr is used to resolve a MAC address in the case of IPv4; the concept of Arp is not supported in IPv6, though.

If the ARP protocol succeeds in resolving the specified address, then the ArpCacheEntries and ArpCache fields of the EFI_PXE_BASE_CODE_MODE structure are updated, and EFI_SUCCESS is returned. If MacAddr is not NULL, the resolved MAC address is placed there as well.

If the PXE Base Code protocol is in the stopped state, then EFI_NOT_STARTED is returned. If the ARP protocol encounters a timeout condition while attempting to resolve an address, then EFI_TIMEOUT is returned. If the Callback Protocol does not return EFI_PXE_BASE_CODE_CALLBACK_STATUS_CONTINUE, then EFI_ABORTED is returned.

Status Codes Returned

<table>
<thead>
<tr>
<th>Status Code</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>EFI_SUCCESS</td>
<td>The IP or MAC address was resolved.</td>
</tr>
<tr>
<td>EFI_INVALID_PARAMETER</td>
<td>One or more of the following conditions was:</td>
</tr>
<tr>
<td></td>
<td>• The This parameter was NULL</td>
</tr>
<tr>
<td></td>
<td>• The This parameter did not point to a valid EFI_PXE_BASE_CODE_PROTOCOL</td>
</tr>
<tr>
<td></td>
<td>structure</td>
</tr>
<tr>
<td></td>
<td>• The IpAddr parameter was NULL</td>
</tr>
<tr>
<td>EFI_DEVICE_ERROR</td>
<td>The network device encountered an error during this operation.</td>
</tr>
<tr>
<td>EFI_NOT_STARTED</td>
<td>The PXE Base Code Protocol is in the stopped state.</td>
</tr>
<tr>
<td>EFI_TIMEOUT</td>
<td>The ARP Protocol encountered a timeout condition.</td>
</tr>
<tr>
<td>EFI_ABORTED</td>
<td>The callback function aborted the ARP Protocol.</td>
</tr>
<tr>
<td>EFI_UNSUPPORTED</td>
<td>When Mode-&gt;UsingIpv6 is TRUE because the Arp is a concept special for IPv4.</td>
</tr>
</tbody>
</table>
**EFI_PXE_BASE_CODE_PROTOCOL.SetParameters()**

**Summary**
Updates the parameters that affect the operation of the PXE Base Code Protocol.

**Prototype**
```c
typedef EFI_STATUS
  (EFIAPIC void EFI_PXE_BASE_CODE_SET_PARAMETERS)
  (IN EFI_PXE_BASE_CODE_PROTOCOL *This,
   IN BOOLEAN *NewAutoArp,  OPTIONAL
   IN BOOLEAN *NewSendGUID,  OPTIONAL
   IN UINT8 *NewTTL,  OPTIONAL
   IN UINT8 *NewToS,  OPTIONAL
   IN BOOLEAN *NewMakeCallback OPTIONAL);
```

**Parameters**
- **This**
  Pointer to the EFI_PXE_BASE_CODE_PROTOCOL instance.
- **NewAutoArp**
  If not NULL, a pointer to a value that specifies whether to replace the current value of AutoARP. If TRUE, ARP packets will be generated as required by the PXE Base Code Protocol. If FALSE, no ARP packets will be generated. In this case, the only mappings that are available are those stored in the ArpCache of the EFI_PXE_BASE_CODE_MODE structure. If there are not enough mappings in the ArpCache to perform a PXE Base Code Protocol service, then the

**Description**
This function sets parameters that affect the operation of the PXE Base Code Protocol. The parameter specified by NewAutoArp is used to control the generation of ARP protocol packets. If NewAutoArp is TRUE, ARP Protocol packets will be generated as required by the PXE Base Code Protocol. If NewAutoArp is FALSE, no ARP Protocol packets will be generated. In this case, the only mappings that are available are those stored in the ArpCache of the EFI_PXE_BASE_CODE_MODE structure. If there are not enough mappings in the ArpCache to perform a PXE Base Code Protocol service, then the
service will fail. This function updates the `AutoArp` field of the `EFI_PXE_BASE_CODE_MODE` structure to `NewAutoArp`.

The `SetParameters()` call must be invoked after a Callback Protocol is installed to enable the use of callbacks.

### Status Codes Returned

<table>
<thead>
<tr>
<th>Code</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>EFI_SUCCESS</td>
<td>The new parameters values were updated.</td>
</tr>
<tr>
<td>EFI_INVALID_PARAMETER</td>
<td>• One or more of the following conditions was <code>TRUE</code>:</td>
</tr>
<tr>
<td></td>
<td>• The <code>This</code> parameter was <code>NULL</code></td>
</tr>
<tr>
<td></td>
<td>• The <code>This</code> parameter did not point to a valid <code>EFI_PXE_BASE_CODE_PROTOCOL</code> structure</td>
</tr>
<tr>
<td></td>
<td>• The <code>NewSendGUID</code> parameter is not <code>NULL</code> and <code>*NewSendGUID</code> is <code>TRUE</code> and a system GUID could not be located</td>
</tr>
<tr>
<td></td>
<td>• The <code>NewMakeCallback</code> parameter is not <code>NULL</code> and <code>*NewMakeCallback</code> is <code>TRUE</code> and an <code>EFI_PXE_BASE_CODE_CALLBACK_PROTOCOL</code> could not be located on the network device handle.</td>
</tr>
<tr>
<td>EFI_NOT_STARTED</td>
<td>The PXE Base Code Protocol is not in the started state.</td>
</tr>
</tbody>
</table>

### `EFI_PXE_BASE_CODE_PROTOCOL.SetStationIp()`

**Summary**

Updates the station IP address and/or subnet mask values of a network device.

**Prototype**

```c
typedef EFI_STATUS
    (EFIAPI *EFI_PXE_BASE_CODE_SET_STATION_IP) (    
        IN EFI_PXE_BASE_CODE_PROTOCOL *This,
        IN EFI_IP_ADDRESS *NewStationIp,   OPTIONAL
        IN EFI_IP_ADDRESS *NewSubnetMask OPTIONAL
    );
```

**Parameters**

- **This**
  - Pointer to the `EFI_PXE_BASE_CODE_PROTOCOL` instance.
- **NewStationIp**
  - Pointer to the new IP address to be used by the network device. If this field is `NULL`, then the `StationIp` address will not be modified.
- **NewSubnetMask**
  - Pointer to the new subnet mask to be used by the network device. If this field is `NULL`, then the `SubnetMask` will not be modified.

**Description**

This function updates the station IP address and/or subnet mask values of a network device.
The `NewStationIp` field is used to modify the network device’s current IP address. If `NewStationIp` is `NULL`, then the current IP address will not be modified. Otherwise, this function updates the `StationIp` field of the `EFI_PXE_BASE_CODE_MODE` structure with `NewStationIp`.

The `NewSubnetMask` field is used to modify the network device’s current subnet mask. If `NewSubnetMask` is `NULL`, then the current subnet mask will not be modified. Otherwise, this function updates the `SubnetMask` field of the `EFI_PXE_BASE_CODE_MODE` structure with `NewSubnetMask`.

### Status Codes Returned

<table>
<thead>
<tr>
<th>Code</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>EFI_SUCCESS</td>
<td>The new station IP address and/or subnet mask were updated.</td>
</tr>
<tr>
<td>EFI_INVALID_PARAMETER</td>
<td>One or more of the following conditions was <code>TRUE</code>:</td>
</tr>
<tr>
<td></td>
<td>• The <code>This</code> parameter was <code>NULL</code></td>
</tr>
<tr>
<td></td>
<td>• The <code>This</code> parameter did not point to a valid <code>EFI_PXE_BASE_CODE_PROTOCOL</code> structure</td>
</tr>
<tr>
<td></td>
<td>• The <code>NewStationIp</code> parameter is not <code>NULL</code> and <code>*NewStationIp</code> is not a valid unicast IP address</td>
</tr>
<tr>
<td></td>
<td>• The <code>NewSubnetMask</code> parameter is not <code>NULL</code> and <code>NewSubnetMask</code> does not contain a valid IP subnet mask</td>
</tr>
<tr>
<td>EFI_NOT_STARTED</td>
<td>The PXE Base Code Protocol is not in the started state.</td>
</tr>
</tbody>
</table>

### EFI_PXE_BASE_CODE_PROTOCOL.SetPackets()

#### Summary

Updates the contents of the cached DHCP and Discover packets.

#### Prototype

```c
typedef
EFI_STATUS
(EFIAPI *EFI_PXE_BASE_CODE_SET_PACKETS) (
    IN EFI_PXE_BASE_CODE_PROTOCOL *This,
    IN BOOLEAN *NewDhcpDiscoverValid, OPTIONAL
    IN BOOLEAN *NewDhcpAckReceived, OPTIONAL
    IN BOOLEAN *NewProxyOfferReceived, OPTIONAL
    IN BOOLEAN *NewPxeDiscoverValid, OPTIONAL
    IN BOOLEAN *NewPxeReplyReceived, OPTIONAL
    IN BOOLEAN *NewPxeBisReplyReceived, OPTIONAL
    IN EFI_PXE_BASE_CODE_PACKET *NewDhcpDiscover, OPTIONAL
    IN EFI_PXE_BASE_CODE_PACKET *NewDhcpAck, OPTIONAL
    IN EFI_PXE_BASE_CODE_PACKET *NewProxyOffer, OPTIONAL
    IN EFI_PXE_BASE_CODE_PACKET *NewPxeDiscover, OPTIONAL
    IN EFI_PXE_BASE_CODE_PACKET *NewPxeReply, OPTIONAL
    IN EFI_PXE_BASE_CODE_PACKET *NewPxeBisReply OPTIONAL
);
```

#### Parameters

- **This**: Pointer to the `EFI_PXE_BASE_CODE_PROTOCOL` instance.
NewDhcpDiscoverValid: Pointer to a value that will replace the current DhcpDiscoverValid field. If NULL, this parameter is ignored.

NewDhcpAckReceived: Pointer to a value that will replace the current DhcpAckReceived field. If NULL, this parameter is ignored.

NewProxyOfferReceived: Pointer to a value that will replace the current ProxyOfferReceived field. If NULL, this parameter is ignored.

NewPxeDiscoverValid: Pointer to a value that will replace the current PxeDiscoverValid field. If NULL, this parameter is ignored.

NewPxeReplyReceived: Pointer to a value that will replace the current PxeReplyReceived field. If NULL, this parameter is ignored.

NewPxeBisReplyReceived: Pointer to a value that will replace the current PxeBisReplyReceived field. If NULL, this parameter is ignored.

NewDhcpDiscover: Pointer to the new cached DHCP Discover packet contents. If NULL, this parameter is ignored.

NewDhcpAck: Pointer to the new cached DHCP Ack packet contents. If NULL, this parameter is ignored.

NewProxyOffer: Pointer to the new cached Proxy Offer packet contents. If NULL, this parameter is ignored.

NewPxeDiscover: Pointer to the new cached PXE Discover packet contents. If NULL, this parameter is ignored.

NewPxeReply: Pointer to the new cached PXE Reply packet contents. If NULL, this parameter is ignored.

NewPxeBisReply: Pointer to the new cached PXE BIS Reply packet contents. If NULL, this parameter is ignored.

Description

The pointers to the new packets are used to update the contents of the cached packets in the EFI_PXE_BASE_CODE_MODE structure.

Status Codes Returned

<table>
<thead>
<tr>
<th>Status Code</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>EFI_SUCCESS</td>
<td>The cached packet contents were updated.</td>
</tr>
<tr>
<td>EFI_INVALID_PARAMETER</td>
<td>• One or more of the following conditions was TRUE:</td>
</tr>
<tr>
<td></td>
<td>• The This parameter was NULL</td>
</tr>
<tr>
<td></td>
<td>The This parameter did not point to a valid</td>
</tr>
<tr>
<td></td>
<td>EFI_PXE_BASE_CODE_PROTOCOL structure.</td>
</tr>
<tr>
<td>EFI_NOT_STARTED</td>
<td>The PXE Base Code Protocol is not in the started state.</td>
</tr>
</tbody>
</table>

24.3.1 Netboot6

For IPv4, PXE drivers typically install a LoadFile protocol on the NIC handle. In the case of supporting both IPv4 and IPv6 where two PXE Base Code and LoadFile protocol instances need be produced, the PXE driver will have to create two child handles and install EFI_LOAD_FILE_PROTOCOL, EFI_SIMPLE_NETWORK_PROTOCOL and PXE_BASE_CODE_PROTOCOL on each child handle. To
distinguish these two child handles, an IP device path node can be appended to the parent device path, for example:

\[
PciRoot(0x0)/Pci(0x19,0x0)/MAC(001320F4B4FF,0x0)/IPv4(...) \ 
\]
\[
PciRoot(0x0)/Pci(0x19,0x0)/MAC(001320F4B4FF,0x0)/IPv6(...) \ 
\]

These two instances allow for the boot manager to decide a preference of IPv6 versus IPv4 since the IETF and other bodies do not speak to this policy choice.

### 24.3.1.1 DHCP6 options for PXE

In IPv4-based PXE, as defined by the rfc2131, rfc2132 and rfc4578, and described by the PXE2.1 specification and the UEFI specification, there are the following PXE related options/fields in DHCPv4 packet:

- siaddr field/ServerAddress option (54) – next server address.
- BootFileName option (67)
- ) – NBP file name.
- BootFileSize option (13)
- – NBP file size.
- ClassIdentifier (60)
- – PXE client tag.
- ClientSystemArchitectureType option (93)
- – client architecture type.
- ClientNetworkInterface Identifier option (94)
- – client network interface identifier.

In IPv6-based PXE, or ‘netboot6’, there are the following PXE related options in the DHCPv6 packet:

- BootFileURL option - OPT_BOOTFILE_URL (59) – next server address and NBP (Network Bootable Program) file name.
- BootFileParameters option
- - OPT_BOOTFILE_PARAM (60) – NBP file size.
- VendorClass option (16)
- – PXE client tag.
- ClientSystemArchitectureType option - OPTION_CLIENT_ARCH_TYPE (61) – client architecture type.
- ClientNetworkInterfaceIdentifier option (62) – client network interface identifier.

The BootFileURL option is used to deliver the next server address or the next server address with NBP file name.

As an example where the next server address delivered only:

As an example where the next server address and BOOTFILE_NAME delivered both:

The BootFileParameters option is used to deliver the NBP file size with the unit of 512-octet blocks. The maximum of the NBP file size is 65535 * 512 bytes.

As an example where the NBP file size is 1600 * 512 bytes:

\[
\begin{align*}
\text{para-len 1} & = 4 \\
\text{parameter 1} & = "1600"
\end{align*}
\]

The VendorClass option is used to deliver the PXE client tag.

As an example where the client architecture is EFI-X64 and the client network interface identifier is UNDI:

\[
\begin{align*}
\text{Enterprise-number} & = (343) \\
\text{Vendor-class-data} & = "\text{PXEClient:Arch:00006:UNDI:003016}"
\end{align*}
\]

#define DUID-UUID 4

The Netboot6 client will use the DUID-UUID to report the platform identifier as part of the netboot6 DHCP options.

**24.3.1.2 IPv6-based PXE boot**

As PXE 2.1 specification describes step-by-step synopsis of the IPv4-based PXE process, Figure 1 describes the corresponding synopsis for netboot6.
24.3.1.2.1 Step 1.

The client multicasts a SOLICIT message to the standard DHCP6 port (547). It contains the following:

- A tag for client UNDI version.
- A tag for the client system architecture.
- A tag for PXE client, Vendor Class data set to “PXEClient:Arch:xxxxx:UNDI:yyyyzzz”.

Figure 24-1  IPv6-based PXE boot
24.3.1.2.2 Step 2.
The DHCP6 or Proxy DHCP6 service responds by sending a ADVERTISE message to the client on the standard DHCP6 reply port (546). If this is a Proxy DHCP6 service, the next server (Boot Server) address is delivered by Boot File URL option. If this is a DHCP6 service, the new assigned client address is delivered by IA option. The extension tags information will be conveyed via the VENDOR OPTS field.

24.3.1.2.3 Steps 3 and 4.
If the client selects an address from a DHCP6 service, then it must complete the standard DHCP6 process by sending a REQUEST for the address back to the service and then waiting for an REPLY from the service.

24.3.1.2.4 Step 5.
The client multicasts a REQUEST message to the Boot Server port 4011, it contains the following:
- A tag for client UNDI version.
- A tag for the client system architecture.
- A tag for PXE client, Vendor Class option, set to "PXEClient:Arch:xxxxx:UNDI:yyyzzz".

24.3.1.2.5 Step 6.
The Boot Server unicasts a REPLY message back to the client on the client port. It contains the following:
- A tag for NBP file name.
- A tag for NBP file size if needed.

24.3.1.2.6 Step 7.
The client requests the NBP file using TFTP (port 69).

24.3.1.2.7 Step 8.
The NBP file, dependent on the client’s CPU architecture, is downloaded into client’s memory.

24.3.1.3 Proxy DHCP6
The netboot6 DHCP6 options may be supplied by the DHCP6 service or a Proxy DHCP6 service. This Proxy DHCP6 service may reside on the same server as the DHCP6 service, or it may be located on a separate server. A Proxy DHCP6 service on the same server as the DHCP6 service is illustrated in Figure 2. In this case, the Proxy DHCP6 service is listening to UDP port (4011), and communication with the Proxy DHCP6 service occurs after completing the standard DHCP6 process. Proxy DHCP6 uses port (4011) because it cannot share port (547) with the DHCP6 service. The netboot6 client knows how to interrogate the Proxy DHCP6 service because the ADVERTISE from the DHCP6 service contains a VendorClass option “PXEClient” tag without a BootFileURL option (including NBP file name). The client will not request option 16 (OPTION_VENDOR_CLASS) in ORO, but server must still reply with "PXEClient" in order to inform the client to start the Proxy DHCPv6 mode. The client will accept just the string "PXEClient" as sufficient, the server need not echo back the entire
Figure 24-2 netboot6 (DHCP6 and ProxyDHCP6 reside on the same server)

Figure 24-3 illustrates the case of a Proxy DHCP6 service and the DHCP6 service on different servers. In this case, the Proxy DHCP6 service listens to UDP port (547) and responds in parallel with DHCP6 service.
24.4 PXE Base Code Callback Protocol

This protocol is a specific instance of the PXE Base Code Callback Protocol that is invoked when the PXE Base Code Protocol is about to transmit, has received, or is waiting to receive a packet. The PXE Base Code Callback Protocol must be on the same handle as the PXE Base Code Protocol.
EFI_PXE_BASE_CODE_CALLBACK_PROTOCOL

Summary
Protocol that is invoked when the PXE Base Code Protocol is about to transmit, has received, or is waiting to receive a packet.

GUID
#define EFI_PXE_BASE_CODE_CALLBACK_PROTOCOL_GUID \
{0x245DCA21,0xFB7B,0x11d3,\ 
{0x8F,0x01,0x00,0xA0, 0xC9,0x69,0x72,0x3B}"

Revision Number
#define EFI_PXE_BASE_CODE_CALLBACK_PROTOCOL_REVISION \
0x00010000

Protocol Interface Structure
typedef struct {
  UINT64                     Revision;
  EFI_PXE_CALLBACK          Callback;
} EFI_PXE_BASE_CODE_CALLBACK_PROTOCOL;

Parameters
Revision          The revision of the EFI_PXE_BASE_CODE_CALLBACK_PROTOCOL. All future revisions must be backwards compatible. If a future revision is not backwards compatible, it is not the same GUID.

Callback          Callback routine used by the PXE Base Code Dhcp(), Discover(), Mtftp(), UdpWrite(), and Arp() functions.

Summary
Callback function that is invoked when the PXE Base Code Protocol is about to transmit, has received, or is waiting to receive a packet.

Prototype
typedef
EFI_PXE_BASE_CODE_CALLBACK_STATUS
(*EFI_PXE_CALLBACK) ( 
  IN EFI_PXE_BASE_CODE_CALLBACK_PROTOCOL  *This, 
  IN EFI_PXE_BASE_CODE_FUNCTION            Function, 
  IN BOOLEAN                                Received, 
  IN UINT32                                  PacketLen, 
  IN EFI_PXE_BASE_CODE_PACKET               *Packet OPTIONAL 
);
Parameters

- **This**: Pointer to the `EFI_PXE_BASE_CODE_PROTOCOL` instance.
- **Function**: The PXE Base Code Protocol function that is waiting for an event.
- **Received**: `TRUE` if the callback is being invoked due to a receive event. `FALSE` if the callback is being invoked due to a transmit event.
- **PacketLen**: The length, in bytes, of `Packet`. This field will have a value of zero if this is a wait for receive event.
- **Packet**: If **Received** is **TRUE**, a pointer to the packet that was just received; otherwise a pointer to the packet that is about to be transmitted. This field will be **NULL** if this is not a packet event.

Related Definitions

```c
typedef enum {
  EFI_PXE_BASE_CODE_CALLBACK_STATUS_FIRST,
  EFI_PXE_BASE_CODE_CALLBACK_STATUS_CONTINUE,
  EFI_PXE_BASE_CODE_CALLBACK_STATUS_ABORT,
  EFI_PXE_BASE_CODE_CALLBACK_STATUS_LAST
} EFI_PXE_BASE_CODE_CALLBACK_STATUS;
```

```c
typedef enum {
  EFI_PXE_BASE_CODE_FUNCTION_FIRST,
  EFI_PXE_BASE_CODE_FUNCTION_DHCP,
  EFI_PXE_BASE_CODE_FUNCTION_DISCOVER,
  EFI_PXE_BASE_CODE_FUNCTION_MTFTP,
  EFI_PXE_BASE_CODE_FUNCTION_UDP_WRITE,
  EFI_PXE_BASE_CODE_FUNCTION_UDP_READ,
  EFI_PXE_BASE_CODE_FUNCTION_ARP,
  EFI_PXE_BASE_CODE_FUNCTION_IGMP,
  EFI_PXE_BASE_CODE_PXE_FUNCTION_LAST
} EFI_PXE_BASE_CODE_FUNCTION;
```

Description

This function is invoked when the PXE Base Code Protocol is about to transmit, has received, or is waiting to receive a packet. Parameters **Function** and **Received** specify the type of event. Parameters **PacketLen** and **Packet** specify the packet that generated the event. If these fields are zero and **NULL** respectively, then this is a status update callback. If the operation specified by **Function** is to continue, then `CALLBACK_STATUS_CONTINUE` should be returned. If the operation specified by **Function** should be aborted, then `CALLBACK_STATUS_ABORT` should be returned. Due to the polling nature of UEFI device drivers, a callback function should not execute for more than 5 ms.
The `SetParameters()` function must be called after a Callback Protocol is installed to enable the use of callbacks.

### 24.5 Boot Integrity Services Protocol

This section defines the Boot Integrity Services (BIS) protocol, which is used to check a digital signature of a data block against a digital certificate for the purpose of an integrity and authorization check. BIS is primarily used by the Preboot Execution Environment (PXE) Base Code protocol `EFI_PXE_BASE_CODE_PROTOCOL` to check downloaded network boot images before executing them. BIS is a UEFI Boot Service Driver, so its services are also available to applications written to this specification until the time of `EFI_BOOT_SERVICES.ExitBootServices()`. More information about BIS can be found in the *Boot Integrity Services Application Programming Interface Version 1.0*.

This section defines the Boot Integrity Services Protocol. This protocol is used to check a digital signature of a data block against a digital certificate for the purpose of an integrity and authorization check.

**EFI_BIS_PROTOCOL**

Summary

The **EFI_BIS_PROTOCOL** is used to check a digital signature of a data block against a digital certificate for the purpose of an integrity and authorization check.

GUID

```c
#define EFI_BIS_PROTOCOL_GUID   \
{0x0b64aab0,0x5429,0x11d4,\ 
 {0x98,0x16,0xa0,0xc9,0x1f,0xad,0xcf}}
```

Protocol Interface Structure

```c
typedef struct _EFI_BIS_PROTOCOL {
  EFI_BIS_INITIALIZE   Initialize;
  EFI_BIS_SHUTDOWN     Shutdown;
  EFI_BIS_FREE         Free;
  EFI_BIS_GET_BOOT_OBJECT_AUTHORIZATION_CERTIFICATE   
    GetBootObjectAuthorizationCertificate;
  EFI_BIS_GET_BOOT_OBJECT_AUTHORIZATION_CHECKFLAG    
    GetBootObjectAuthorizationCheckFlag;
  EFI_BIS_GET_BOOT_OBJECT_AUTHORIZATION_UPDATE_TOKEN 
    GetBootObjectAuthorizationUpdateToken;
  EFI_BIS_GET_SIGNATURE_INFO                         
    GetSignatureInfo;
  EFI_BIS_UPDATE_BOOT_OBJECT_AUTHORIZATION          
    UpdateBootObjectAuthorization;
  EFI_BIS_VERIFY_BOOT_OBJECT                         
    VerifyBootObject;
  EFI_BIS_VERIFY_OBJECT_WITH_CREDENTIAL              
    VerifyObjectWithCredential;
} EFI_BIS_PROTOCOL;
```
Parameters

**Initialize**
Initializes an application instance of the **EFI_BIS** protocol, returning a handle for the application instance. Other functions in the **EFI_BIS** protocol require a valid application instance handle obtained from this function. See the **Initialize()** function description.

**Shutdown**
Ends the lifetime of an application instance of the **EFI_BIS** protocol, invalidating its application instance handle. The application instance handle may no longer be used in other functions in the **EFI_BIS** protocol. See the **Shutdown()** function description.

**Free**
Frees memory structures allocated and returned by other functions in the **EFI_BIS** protocol. See the **Free()** function description.

**GetBootObjectAuthorizationCertificate**
Retrieves the current digital certificate (if any) used by the **EFI_BIS** protocol as the source of authorization for verifying boot objects and altering configuration parameters. See the **GetBootObjectAuthorizationCertificate()** function description.

**GetBootObjectAuthorizationCheckFlag**
Retrieves the current setting of the authorization check flag that indicates whether or not authorization checks are required for boot objects. See the **GetBootObjectAuthorizationCheckFlag()** function description.

**GetBootObjectAuthorizationUpdateToken**
Retrieves an uninterpreted token whose value gets included and signed in a subsequent request to alter the configuration parameters, to protect against attempts to “replay” such a request. See the **GetBootObjectAuthorizationUpdateToken()** function description.

**GetSignatureInfo**
Retrieves information about the digital signature algorithms supported and the identity of the installed authorization certificate, if any. See the **GetSignatureInfo()** function description.

**UpdateBootObjectAuthorization**
Requests that the configuration parameters be altered by installing or removing an authorization certificate or changing the setting of the check flag. See the **UpdateBootObjectAuthorization()** function description.

**VerifyBootObject**
Verifies a boot object according to the supplied digital signature and the current authorization certificate and check flag setting. See the **VerifyBootObject()** function description.

**VerifyObjectWithCredential**
Verifies a data object according to a supplied digital signature and a supplied digital certificate. See the **VerifyObjectWithCredential()** function description.
Description
The EFI_BIS_PROTOCOL provides a set of functions as defined in this section. There is no physical device associated with these functions, however, in the context of UEFI every protocol operates on a device. Accordingly, BIS installs and operates on a single abstract device that has only a software representation.


EFI_BIS_PROTOCOL.Initialize()

Summary
Initializes the BIS service, checking that it is compatible with the version requested by the caller. After this call, other BIS functions may be invoked.

Prototype

```c
typedef EFI_STATUS
  (EFIAPI *EFI_BIS_INITIALIZE)(
   IN   EFI_BIS_PROTOCOL    *This,
   OUT  BIS_APPLICATION_HANDLE *AppHandle,
   IN OUT EFI_BIS_VERSION    *InterfaceVersion,
   IN   EFI_BIS_DATA        *TargetAddress
  );
```

Parameters

This
A pointer to the EFI_BIS_PROTOCOL object. The protocol implementation may rely on the actual pointer value and object location, so the caller must not copy the object to a new location.

AppHandle
The function writes the new BIS_APPLICATION_HANDLE if successful, otherwise it writes NULL. The caller must eventually destroy this handle by calling Shutdown(). Type BIS_APPLICATION_HANDLE is defined in “Related Definitions” below.

InterfaceVersion
On input, the caller supplies the major version number of the interface version desired. The minor version number supplied on input is ignored since interface compatibility is determined solely by the major version number. On output, both the major and minor version numbers are updated with the major and minor version numbers of the interface (and underlying implementation). This update is done whether or not the initialization was successful. Type EFI_BIS_VERSION is defined in “Related Definitions” below.

TargetAddress
Indicates a network or device address of the BIS platform to connect to. Local-platform BIS implementations require that the caller sets TargetAddress.Data to NULL, but otherwise ignores this parameter. BIS implementations that redirect calls to an agent at a remote address must define their own format and interpretation of this parameter outside the scope of this document. For all implementations, if the TargetAddress is an unsupported value,
the function fails with the error **EFI_UNSUPPORTED**. Type **EFI_BIS_DATA** is defined in “Related Definitions” below.

**Related Definitions**

```c
//*******************************************************************************
//  BIS_APPLICATION_HANDLE
//*******************************************************************************
typedef VOID *BIS_APPLICATION_HANDLE;
```

This type is an opaque handle representing an initialized instance of the BIS interface. A **BIS_APPLICATION_HANDLE** value is returned by the **Initialize()** function as an “out” parameter. Other BIS functions take a **BIS_APPLICATION_HANDLE** as an “in” parameter to identify the BIS instance.

```c
//*******************************************************************************
//  EFI_BIS_VERSION
//*******************************************************************************
typedef struct _EFI_BIS_VERSION {
  UINT32 Major;
  UINT32 Minor;
} EFI_BIS_VERSION;
```

**Major**

This describes the major BIS version number. The major version number defines version compatibility. That is, when a new version of the BIS interface is created with new capabilities that are not available in the previous interface version, the major version number is increased.

**Minor**

This describes a minor BIS version number. This version number is increased whenever a new BIS implementation is built that is fully interface compatible with the previous BIS implementation. This number may be reset when the major version number is increased.

This type represents a version number of the BIS interface. This is used as an “in out” parameter of the **Initialize()** function for a simple form of negotiation of the BIS interface version between the caller and the BIS implementation.

```c
//*******************************************************************************
//  EFI_BIS_VERSION predefined values
//  Use these values to initialize EFI_BIS_VERSION.Major and to interpret results of Initialize.
//*******************************************************************************
#define BIS_CURRENT_VERSION_MAJOR   BIS_VERSION_1
#define BIS_VERSION_1               1
```

These C preprocessor macros supply values for the major version number of an **EFI_BIS_VERSION**. At the time of initialization, a caller supplies a value to request a BIS interface version. On return, the (IN OUT) parameter is over-written with the actual version of the interface.
typedef struct _EFI_BIS_DATA {
    UINT32   Length;
    UINT8    *Data;
} EFI_BIS_DATA;

The length of the data buffer in bytes.
A pointer to the raw data buffer.

This type defines a structure that describes a buffer. BIS uses this type to pass back and forth most large objects such as digital certificates, strings, etc. Several of the BIS functions allocate a EFI_BIS_DATA* and return it as an “out” parameter. The caller must eventually free any allocated EFI_BIS_DATA* using the Free() function.

Description
This function must be the first BIS function invoked by an application. It passes back a BIS_APPLICATION_HANDLE value that must be used in subsequent BIS functions. The handle must be eventually destroyed by a call to the Shutdown() function, thus ending that handle’s lifetime. After the handle is destroyed, BIS functions may no longer be called with that handle value. Thus all other BIS functions may only be called between a pair of Initialize() and Shutdown() functions.

There is no penalty for calling Initialize() multiple times. Each call passes back a distinct handle value. Each distinct handle must be destroyed by a distinct call to Shutdown(). The lifetimes of handles created and destroyed with these functions may be overlapped in any way.
Status Codes Returned

<table>
<thead>
<tr>
<th>Status Code</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>EFI_SUCCESS</td>
<td>The function completed successfully.</td>
</tr>
<tr>
<td>EFI_INCOMPATIBLE_VERSION</td>
<td>The InterfaceVersion.Major requested by the caller was not compatible with the interface version of the implementation. The InterfaceVersion.Major has been updated with the current interface version.</td>
</tr>
<tr>
<td>EFI_UNSUPPORTED</td>
<td>This is a local-platform implementation and TargetAddress.Data was not NULL, or TargetAddress.Data was any other value that was not supported by the implementation.</td>
</tr>
<tr>
<td>EFI_OUT_OF_RESOURCES</td>
<td>The function failed due to lack of memory or other resources.</td>
</tr>
<tr>
<td>EFI_DEVICE_ERROR</td>
<td>The function encountered an unexpected internal failure while initializing a cryptographic software module, or No cryptographic software module with compatible version was found, or A resource limitation was encountered while using a cryptographic software module.</td>
</tr>
<tr>
<td>EFI_INVALID_PARAMETER</td>
<td>The This parameter supplied by the caller is NULL or does not reference a valid EFI_BIS_PROTOCOL object, or The AppHandle parameter supplied by the caller is NULL or an invalid memory reference, or The InterfaceVersion parameter supplied by the caller is NULL or an invalid memory reference, or The TargetAddress parameter supplied by the caller is NULL or an invalid memory reference.</td>
</tr>
</tbody>
</table>

**EFI_BIS_PROTOCOL.Shutdown()**

**Summary**

Shuts down an application’s instance of the BIS service, invalidating the application handle. After this call, other BIS functions may no longer be invoked using the application handle value.

**Prototype**

```c
typedef
EFI_STATUS
(EFIAPI *EFI_BIS_SHUTDOWN)(
    IN BIS_APPLICATION_HANDLE AppHandle
);
```

**Parameters**

- **AppHandle** An opaque handle that identifies the caller’s instance of initialization of the BIS service. Type BIS_APPLICATION_HANDLE is defined in the Initialize() function description.
Description
This function shuts down an application’s instance of the BIS service, invalidating the application handle. After this call, other BIS functions may no longer be invoked using the application handle value.

This function must be paired with a preceding successful call to the Initialize() function. The lifetime of an application handle extends from the time the handle was returned from Initialize() until the time the handle is passed to Shutdown(). If there are other remaining handles whose lifetime is still active, they may still be used in calling BIS functions.

The caller must free all memory resources associated with this AppHandle that were allocated and returned from other BIS functions before calling Shutdown(). Memory resources are freed using the Free() function. Failure to free such memory resources is a caller error, however, this function does not return an error code under this circumstance. Further attempts to access the outstanding memory resources cause unspecified results.

Status Codes Returned

<table>
<thead>
<tr>
<th>Status Code</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>EFI_SUCCESS</td>
<td>The function completed successfully.</td>
</tr>
<tr>
<td>EFI_NO_MAPPING</td>
<td>The AppHandle parameter is not or is no longer a valid application instance handle associated with the EFI_BIS protocol.</td>
</tr>
<tr>
<td>EFI_DEVICE_ERROR</td>
<td>The function encountered an unexpected internal error while returning resources associated with a cryptographic software module, or The function encountered an internal error while trying to shut down a cryptographic software module.</td>
</tr>
<tr>
<td>EFI_OUT_OF_RESOURCES</td>
<td>The function failed due to lack of memory or other resources.</td>
</tr>
</tbody>
</table>

EFI_BIS_PROTOCOL.Free()

Summary
Frees memory structures allocated and returned by other functions in the EFI_BIS protocol.

Prototype
define

typedef

EFI_STATUS

(EIFIAPI *EFI_BIS_FREE)(
    IN BIS_APPLICATION_HANDLE AppHandle,
    IN EFI_BIS_DATA      *ToF


); Parameters

AppHandle An opaque handle that identifies the caller’s instance of initialization of the BIS service. Type BIS_APPLICATION_HANDLE is defined in the Initialize() function description.

ToFree An EFI_BIS_DATA* and associated memory block to be freed. This EFI_BIS_DATA* must have been allocated by one of the other BIS functions. Type EFI_BIS_DATA is defined in the Initialize() function description.
Description
This function deallocates an `EFI_BIS_DATA*` and associated memory allocated by one of the other BIS functions.

Callers of other BIS functions that allocate memory in the form of an `EFI_BIS_DATA*` must eventually call this function to deallocate the memory before calling the `Shutdown()` function for the application handle under which the memory was allocated. Failure to do so causes unspecified results, and the continued correct operation of the BIS service cannot be guaranteed.

Status Codes Returned

<table>
<thead>
<tr>
<th>Status Code</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>EFI_SUCCESS</td>
<td>The function completed successfully.</td>
</tr>
<tr>
<td>EFI_NO_MAPPING</td>
<td>The <code>AppHandle</code> parameter is not or is no longer a valid application instance handle associated with the EFI_BIS protocol.</td>
</tr>
<tr>
<td>EFI_INVALID_PARAMETER</td>
<td>The <code>ToFree</code> parameter is not or is no longer a memory resource associated with this <code>AppHandle</code>.</td>
</tr>
<tr>
<td>EFI_OUT_OF_RESOURCES</td>
<td>The function failed due to lack of memory or other resources.</td>
</tr>
</tbody>
</table>

`EFI_BIS_PROTOCOL.GetBootObjectAuthorizationCertificate()`

Summary
Retrieves the certificate that has been configured as the identity of the organization designated as the source of authorization for signatures of boot objects.

Prototype

```c
typedef
EFI_STATUS
(EFIAPI *EFI_BIS_GET_BOOT_OBJECT_AUTHORIZATION_CERTIFICATE)(
    IN BIS_APPLICATION_Handle AppHandle,
    OUT EFI_BIS_DATA **Certificate
);
```

Parameters

- **AppHandle**
  An opaque handle that identifies the caller’s instance of initialization of the BIS service. Type `BIS_APPLICATION_HANDLE` is defined in the `Initialize()` function description.

- **Certificate**
  The function writes an allocated `EFI_BIS_DATA*` containing the Boot Object Authorization Certificate object. The caller must eventually free the memory allocated by this function using the function `Free()`. Type `EFI_BIS_DATA` is defined in the `Initialize()` function description.

Description
This function retrieves the certificate that has been configured as the identity of the organization designated as the source of authorization for signatures of boot objects.
Status CodesReturned

<table>
<thead>
<tr>
<th>Code</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>EFI_SUCCESS</td>
<td>The function completed successfully.</td>
</tr>
<tr>
<td>EFI_NO_MAPPING</td>
<td>The AppHandle parameter is not or is no longer a valid application instance handle associated with the EFI_BIS protocol.</td>
</tr>
<tr>
<td>EFI_NOT_FOUND</td>
<td>There is no Boot Object Authorization Certificate currently installed.</td>
</tr>
<tr>
<td>EFI_OUT_OF_RESOURCES</td>
<td>The function failed due to lack of memory or other resources.</td>
</tr>
<tr>
<td>EFI_INVALID_PARAMETER</td>
<td>The Certificate parameter supplied by the caller is NULL or an invalid memory reference.</td>
</tr>
</tbody>
</table>

**EFI_BIS_PROTOCOL.GetBootObjectAuthorizationCheckFlag()**

**Summary**
Retrieves the current status of the Boot Authorization Check Flag.

**Prototype**
```c
typedef EFI_STATUS (EFIAPIS *EFI_BIS_GET_BOOT_OBJECT_AUTHORIZATION_CHECKFLAG)(
    IN BIS_APPLICATION_HANDLE AppHandle,
    OUT BOOLEAN *CheckIsRequired
);
```

**Parameters**
- **AppHandle**
  - An opaque handle that identifies the caller’s instance of initialization of the BIS service. Type `BIS_APPLICATION_HANDLE` is defined in the Initialize() function description.
- **CheckIsRequired**
  - The function writes the value **TRUE** if a Boot Authorization Check is currently required on this platform, otherwise the function writes **FALSE**.

**Description**
This function retrieves the current status of the Boot Authorization Check Flag (in other words, whether or not a Boot Authorization Check is currently required on this platform).
Status Codes Returned

<table>
<thead>
<tr>
<th>Status Code</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>EFI_SUCCESS</td>
<td>The function completed successfully.</td>
</tr>
<tr>
<td>EFI_NO_MAPPING</td>
<td>The AppHandle parameter is not or is no longer a valid application instance handle associated with the EFI_BIS protocol.</td>
</tr>
<tr>
<td>EFI_OUT_OF_RESOURCES</td>
<td>The function failed due to lack of memory or other resources.</td>
</tr>
<tr>
<td>EFI_INVALID_PARAMETER</td>
<td>The CheckIsRequired parameter supplied by the caller is NULL or an invalid memory reference.</td>
</tr>
</tbody>
</table>

**EFI_BIS_PROTOCOL.GetBootObjectAuthorizationUpdateToken()**

**Summary**
Retrieves a unique token value to be included in the request credential for the next update of any parameter in the Boot Object Authorization set (Boot Object Authorization Certificate and Boot Authorization Check Flag).

**Prototype**
```c
typedef EFI_STATUS (EFIAPI *EFI_BIS_GET_BOOT_OBJECT_AUTHORIZATION_UPDATE_TOKEN)(
    IN BIS_APPLICATION_HANDLE AppHandle,
    OUT EFI_BIS_DATA **UpdateToken
);
```

**Parameters**
- **AppHandle**
  An opaque handle that identifies the caller’s instance of initialization of the BIS service. Type BIS_APPLICATION_HANDLE is defined in the Initialize() function description.
- **UpdateToken**
  The function writes an allocated EFI_BIS_DATA* containing the new unique update token value. The caller must eventually free the memory allocated by this function using the function Free(). Type EFI_BIS_DATA is defined in the Initialize() function description.

**Description**
This function retrieves a unique token value to be included in the request credential for the next update of any parameter in the Boot Object Authorization set (Boot Object Authorization Certificate and Boot Authorization Check Flag). The token value is unique to this platform, parameter set, and instance of parameter values. In particular, the token changes to a new unique value whenever any parameter in this set is changed.
Status Codes Returned

<table>
<thead>
<tr>
<th>Code</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>EFI_SUCCESS</td>
<td>The function completed successfully.</td>
</tr>
<tr>
<td>EFI_NO_MAPPING</td>
<td>The <code>AppHandle</code> parameter is not or is no longer a valid application instance handle associated with the EFI_BIS protocol.</td>
</tr>
<tr>
<td>EFI_OUT_OF_RESOURCES</td>
<td>The function failed due to lack of memory or other resources.</td>
</tr>
<tr>
<td>EFI_DEVICE_ERROR</td>
<td>The function encountered an unexpected internal error in a cryptographic software module.</td>
</tr>
<tr>
<td>EFI_INVALID_PARAMETER</td>
<td>The <code>UpdateToken</code> parameter supplied by the caller is <code>NULL</code> or an invalid memory reference.</td>
</tr>
</tbody>
</table>

**EFI_BIS_PROTOCOL.GetSignatureInfo()**

**Summary**

Retrieves a list of digital certificate identifier, digital signature algorithm, hash algorithm, and key-length combinations that the platform supports.

**Prototype**

```c
typedef
EFI_STATUS
(EFI_API *EFI_BIS_GET_SIGNATURE_INFO)(
    IN BIS_APPLICATION_HANDLE AppHandle,
    OUT EFI_BIS_DATA **SignatureInfo
);
```

**Parameters**

- **AppHandle**
  - An opaque handle that identifies the caller’s instance of initialization of the BIS service. Type `BIS_APPLICATION_HANDLE` is defined in the [Initialize()] function description.

- **SignatureInfo**
  - The function writes an allocated `EFI_BIS_DATA*` containing the array of `EFI_BIS_SIGNATURE_INFO` structures representing the supported digital certificate identifier, algorithm, and key length combinations. The caller must eventually free the memory allocated by this function using the function `Free()`. Type `EFI_BIS_DATA` is defined in the [Initialize()] function description. Type `EFI_BIS_SIGNATURE_INFO` is defined in “Related Definitions” below.
Related Definitions

typedef struct _EFI_BIS_SIGNATURE_INFO {
    BIS_CERT_ID   CertificateID;
    BIS_ALG_ID    AlgorithmID;
    UINT16        KeyLength;
} EFI_BIS_SIGNATURE_INFO;

CertificateID
A shortened value identifying the platform’s currently configured Boot Object Authorization Certificate, if one is currently configured. The shortened value is derived from the certificate as defined in the Related Definition for BIS_CERT_ID below. If there is no certificate currently configured, the value is one of the reserved BIS_CERT_ID XXX values defined below. Type BIS_CERT_ID and its predefined reserved values are defined in “Related Definitions” below.

AlgorithmID
A predefined constant representing a particular digital signature algorithm. Often this represents a combination of hash algorithm and encryption algorithm, however, it may also represent a standalone digital signature algorithm. Type BIS_ALG_ID and its permitted values are defined in “Related Definitions” below.

KeyLength
The length of the public key, in bits, supported by this digital signature algorithm.

This type defines a digital certificate, digital signature algorithm, and key-length combination that may be supported by the BIS implementation. This type is returned by GetSignatureInfo() to describe the combination(s) supported by the implementation.

#define BIS_GET_SIGINFO_COUNT(BisDataPtr) \
    ((BisDataPtr)->Length/sizeof(EFI_BIS_SIGNATURE_INFO))

BisDataPtr Supplies the pointer to the target EFI_BIS_DATA structure.
(return value) The number of EFI_BIS_SIGNATURE_INFO elements contained in the array.

This macro computes how many EFI_BIS_SIGNATURE_INFO elements are contained in an EFI_BIS_DATA structure returned from GetSignatureInfo(). The number returned is the count of items in the list of supported digital certificate, digital signature algorithm, and key-length combinations.
//******************************************************************************************
// BIS_GET_SIGINFO_ARRAY macro
// Produces a EFI_BIS_SIGNATURE_INFO* from a given
// EFI_BIS_DATA*.
//******************************************************************************************
#define BIS_GET_SIGINFO_ARRAY(BisDataPtr) 
  ((EFI_BIS_SIGNATURE_INFO*)(BisDataPtr)->Data)

BisDataPtr Supplies the pointer to the target EFI_BIS_DATA structure.

(return value) The pointer to the EFI_BIS_SIGNATURE_INFO array, cast as an
  EFI_BIS_SIGNATURE_INFO*.

This macro returns a pointer to the EFI_BIS_SIGNATURE_INFO array contained in an EFI_BIS_DATA
structure returned from GetSignatureInfo() representing the list of supported digital certificate,
digital signature algorithm, and key-length combinations.

//******************************************************************************************
// BIS_CERT_ID
//******************************************************************************************
typedef UINT32 BIS_CERT_ID;

This type represents a shortened value that identifies the platform’s currently configured Boot Object
Authorization Certificate. The value is the first four bytes, in “little-endian” order, of the SHA-1 hash of
the certificate, except that the most-significant bits of the second and third bytes are reserved, and must
be set to zero regardless of the outcome of the hash function. This type is included in the array of values
returned from the GetSignatureInfo() function to indicate the required source of a signature for a
boot object or a configuration update request. There are a few predefined reserved values with special
meanings as described below.

//******************************************************************************************
// BIS_CERT_ID predefined values
// Currently defined values for EFI_BIS_SIGNATURE_INFO.
// CertificateId.
//******************************************************************************************
#define BIS_CERT_ID_DSA   BIS_ALG_DSA   //CSSM_ALGID_DSA
#define BIS_CERT_ID_RSA_MD5 BIS_ALG_RSA_MD5 //CSSM_ALGID_MD5_WITH_RSA

These C preprocessor symbols provide values for the BIS_CERT_ID type. These values are used when
the platform has no configured Boot Object Authorization Certificate. They indicate the signature
algorithm that is supported by the platform. Users must be careful to avoid constructing Boot Object
Authorization Certificates that transform to BIS_CERT_ID values that collide with these predefined
values or with the BIS_CERT_ID values of other Boot Object Authorization Certificates they use.
//*******************************************************
// BIS_CERT_ID_MASK
// The following is a mask value that gets applied to the
// truncated hash of a platform Boot Object Authorization
// Certificate to create the CertificateId. A CertificateId
// must not have any bits set to the value 1 other than bits in
// this mask.
//*******************************************************
#define BIS_CERT_ID_MASK (0xFF7F7FFF)

This C preprocessor symbol may be used as a bit-wise “AND” value to transform the first four bytes (in
little-endian order) of a SHA-1 hash of a certificate into a certificate ID with the “reserved” bits properly
set to zero.

//*******************************************************
// BIS_ALG_ID
//*******************************************************
typedef UINT16         BIS_ALG_ID;

This type represents a digital signature algorithm. A digital signature algorithm is often composed of a
particular combination of secure hash algorithm and encryption algorithm. This type also allows for
digital signature algorithms that cannot be decomposed. Predefined values for this type are as defined
below.

//*******************************************************
// BIS_ALG_ID predefined values
// Currently defined values for EFI_BIS_SIGNATURE_INFO.
// AlgorithmId. The exact numeric values come from “Common
// Data Security Architecture (CDSA) Specification.”
//*******************************************************
#define BIS_ALG_DSA   (41)  //CSSM_ALGID_DSA
#define BIS_ALG_RSA_MD5 (42)  //CSSM_ALGID_MD5_WITH_RSA

These values represent the two digital signature algorithms predefined for BIS. Each implementation of
BIS must support at least one of these digital signature algorithms. Values for the digital signature
algorithms are chosen by an industry group known as The Open Group. Developers planning to support
additional digital signature algorithms or define new digital signature algorithms should refer to The
Open Group for interoperable values to use.

Description
This function retrieves a list of digital certificate identifier, digital signature algorithm, hash algorithm,
and key-length combinations that the platform supports. The list is an array of (certificate id, algorithm
id, key length) triples, where the certificate id is derived from the platform’s Boot Object Authorization
Certificate as described in the Related Definition for BIS_CERT_ID above, the algorithm id represents
the combination of signature algorithm and hash algorithm, and the key length is expressed in bits. The
number of array elements can be computed using the Length field of the retrieved EFI_BIS_DATA*. 
The retrieved list is in order of preference. A digital signature algorithm for which the platform has a currently configured Boot Object Authorization Certificate is preferred over any digital signature algorithm for which there is not a currently configured Boot Object Authorization Certificate. Thus the first element in the list has a `CertificateID` representing a Boot Object Authorization Certificate if the platform has one configured. Otherwise the `CertificateID` of the first element in the list is one of the reserved values representing a digital signature algorithm.

### Status Codes Returned

<table>
<thead>
<tr>
<th>Status Code</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>EFI_SUCCESS</td>
<td>The function completed successfully.</td>
</tr>
<tr>
<td>EFI_NO_MAPPING</td>
<td>The <code>AppHandle</code> parameter is not or is no longer a valid application instance handle associated with the EFI_BIS protocol.</td>
</tr>
<tr>
<td>EFI_OUT_OF_RESOURCES</td>
<td>The function failed due to lack of memory or other resources.</td>
</tr>
<tr>
<td>EFI_DEVICE_ERROR</td>
<td>The function encountered an unexpected internal error in a cryptographic software module, or The function encountered an unexpected internal consistency check failure (possible corruption of stored Boot Object Authorization Certificate).</td>
</tr>
<tr>
<td>EFI_INVALID_PARAMETER</td>
<td>The <code>SignatureInfo</code> parameter supplied by the caller is NULL or an invalid memory reference.</td>
</tr>
</tbody>
</table>

### EFI_BIS_PROTOCOL.UpdateBootObjectAuthorization()

**Summary**

Updates one of the configurable parameters of the Boot Object Authorization set (Boot Object Authorization Certificate or Boot Authorization Check Flag).

**Prototype**

```c
typedef EFI_STATUS
(EFIAPI *EFI_BIS_UPDATE_BOOT_OBJECT_AUTHORIZATION)(
    IN BIS_APPLICATION_HANDLE AppHandle,
    IN EFI_BIS_DATA *RequestCredential,
    OUT EFI_BIS_DATA **NewUpdateToken
);
```

**Parameters**

- **AppHandle**
  
  An opaque handle that identifies the caller’s instance of initialization of the BIS service. Type `BIS_APPLICATION_HANDLE` is defined in the `Initialize()` function description.

- **RequestCredential**
  
  This is a Signed Manifest with embedded attributes that carry the details of the requested update. The required syntax of the Signed Manifest is described in the Related Definition for Manifest Syntax below. The key used to sign the request credential must be the private key corresponding to the public key in the platform’s configured Boot Object Authorization Certificate. Authority to
update parameters in the Boot Object Authorization set cannot be delegated.

If there is no Boot Object Authorization Certificate, the request credential may be signed with any private key. In this case, this function interacts with the user in a platform-specific way to determine whether the operation should succeed. Type EFI_BIS_DATA is defined in the Initialize() function description.

NewUpdateToken

The function writes an allocated EFI_BIS_DATA* containing the new unique update token value. The caller must eventually free the memory allocated by this function using the function Free(). Type EFI_BIS_DATA is defined in the Initialize() function description.

Related Definitions

//**********************************************************
// Manifest Syntax
//**********************************************************

The Signed Manifest consists of three parts grouped together into an Electronic Shrink Wrap archive as described in [SM spec]: a manifest file, a signer’s information file, and a signature block file. These three parts, along with examples are described in the following sections. In these examples, text in parentheses is a description of the text that would appear in the signed manifest. Text outside of parentheses must appear exactly as shown. Also note that manifest files and signer’s information files must conform to a 72-byte line-length limit. Continuation lines (lines beginning with a single “space” character) are used for lines longer than 72 bytes. The examples given here follow this rule for continuation lines.

Note that the manifest file and signer’s information file parts of a Signed Manifest are ASCII text files. In cases where these files contain a base-64 encoded string, the string is an ASCII string before base-64 encoding.

//**********************************************************
// Manifest File Example
//**********************************************************

The manifest file must include a section referring to a memory-type data object with the reserved name as shown in the example below. This data object is a zero-length object whose sole purpose in the manifest is to serve as a named collection point for the attributes that carry the details of the requested update. The attributes are also contained in the manifest file. An example manifest file is shown below.

Manifest-Version: 2.0
ManifestPersistentId: (base-64 representation of a unique GUID)

Name: memory:UpdateRequestParameters
Digest-Algorithms: SHA-1
SHA-1-Digest: (base-64 representation of a SHA-1 digest of zero-length buffer)
X-Intel-BIS-ParameterSet: (base-64 representation of
A line-by-line description of this manifest file is as follows.

**Manifest-Version: 2.0**

This is a standard header line that all signed manifests have. It must appear exactly as shown.

**ManifestPersistentId: (base-64 representation of a unique GUID)**

The left-hand string must appear exactly as shown. The right-hand string must be a unique GUID for every manifest file created. The Win32 function UuidCreate() can be used for this on Win32 systems. The GUID is a binary value that must be base-64 encoded. Base-64 is a simple encoding scheme for representing binary values that uses only printing characters. Base-64 encoding is described in [BASE-64].

**Name: memory:UpdateRequestParameters**

This identifies the manifest section that carries a dummy zero-length data object serving as the collection point for the attribute values appearing later in this manifest section (lines prefixed with “X-Intel-BIS-”). The string “memory:UpdateRequestParameters” must appear exactly as shown.

**Digest-Algorithms: SHA-1**

This enumerates the digest algorithms for which integrity data is included for the data object. These are required even though the data object is zero-length. For systems with DSA signing, SHA-1 hash, and 1024-bit key length, the digest algorithm must be “SHA-1.” For systems with RSA signing, MD5 hash, and 512-bit key length, the digest algorithm must be “MD5.” Multiple algorithms can be specified as a whitespace-separated list. For every digest algorithm XXX listed, there must also be a corresponding XXX-Digest line.

**SHA-1-Digest: (base-64 representation of a SHA-1 digest of zero-length buffer)**

Gives the corresponding digest value for the dummy zero-length data object. The value is base-64 encoded. Note that for both MD5 and SHA-1, the digest value for a zero-length data object is not zero.

**X-Intel-BIS-ParameterSet: (base-64 representation of BootObjectAuthorizationSetGUID)**

A named attribute value that distinguishes updates of BIS parameters from updates of other parameters. The left-hand attribute-name keyword must appear exactly as shown. The GUID value for the right-hand side is always the same, and can be found under the preprocessor symbol `BOOT_OBJECT_AUTHORIZATION_PARAMSET_GUIDVALUE`. The representation inserted into the manifest is base-64 encoded.

Note the “X-Intel-BIS-” prefix on this and the following attributes. The “X-” part of the prefix was chosen to avoid collisions with future reserved keywords defined by future versions of the signed
manifest specification. The “Intel-BIS-” part of the prefix was chosen to avoid collisions with other user-defined attribute names within the user-defined attribute name space.

X-Intel-BIS-ParameterSetToken: (base-64 representation of the current update token)

A named attribute value that makes this update of BIS parameters different from any other on the same target platform. The left-hand attribute-name keyword must appear exactly as shown. The value for the right-hand side is generally different for each update-request manifest generated. The value to be base-64 encoded is retrieved through the functions GetBootObjectAuthorizationUpdateToken() or UpdateBootObjectAuthorization().

X-Intel-BIS-ParameterId: (base-64 representation of “BootObjectAuthorizationCertificate” or “BootAuthorizationCheckFlag”)

A named attribute value that indicates which BIS parameter is to be updated. The left-hand attribute-name keyword must appear exactly as shown. The value for the right-hand side is the base-64 encoded representation of one of the two strings shown.

X-Intel-BIS-ParameterValue: (base-64 representation of certificate or single-byte boolean flag)

A named attribute value that indicates the new value to be set for the indicated parameter. The left-hand attribute-name keyword must appear exactly as shown. The value for the right-hand side is the appropriate base-64 encoded new value to be set. In the case of the Boot Object Authorization Certificate, the value is the new digital certificate raw data. A zero-length value removes the certificate altogether. In the case of the Boot Authorization Check Flag, the value is a single-byte Boolean value, where a nonzero value “turns on” the check and a zero value “turns off” the check.

//**********************************************************
// Signer’s Information File Example
//**********************************************************

The signer’s information file must include a section whose name matches the reserved data object section name of the section in the Manifest file. This section in the signer’s information file carries the integrity data for the attributes in the corresponding section in the manifest file. An example signer’s information file is shown below.

Signature-Version: 2.0
SignerInformationPersistentId: (base-64 representation of a unique GUID)
SignerInformationName: BIS_UpdateManifestSignerInfoName

Name: memory:UpdateRequestParameters
Digest-Algorithms: SHA-1
SHA-1-Digest: (base-64 representation of a SHA-1 digest of the corresponding manifest section)

A line-by-line description of this signer’s information file is as follows.

Signature-Version: 2.0
This is a standard header line that all signed manifests have. It must appear exactly as shown.

**SignerInformationPersistentId:** (base-64 representation of a unique GUID)

The left-hand string must appear exactly as shown. The right-hand string must be a unique GUID for every signer’s information file created. The Win32 function UuidCreate() can be used for this on Win32 systems. The GUID is a binary value that must be base-64 encoded. Base-64 is a simple encoding scheme for representing binary values that uses only printing characters. Base-64 encoding is described in [BASE-64].

**SignerInformationName:** BIS_UpdateManifestSignerInfoName

The left-hand string must appear exactly as shown. The right-hand string must appear exactly as shown.

**Name:** memory:UpdateRequestParameters

This identifies the section in the signer’s information file corresponding to the section with the same name in the manifest file described earlier. The string “memory:UpdateRequestParameters” must appear exactly as shown.

**Digest-Algorithms:** SHA-1

This enumerates the digest algorithms for which integrity data is included for the corresponding manifest section. Strings identifying digest algorithms are the same as in the manifest file. The digest algorithms specified here must match those specified in the manifest file. For every digest algorithm XXX listed, there must also be a corresponding **XXX-Digest** line.

**SHA-1-Digest:** (base-64 representation of a SHA-1 digest of the corresponding manifest section)

Gives the corresponding digest value for the corresponding manifest section. The value is base-64 encoded. Note that for the purpose of computing the hash of the manifest section, the manifest section starts at the beginning of the opening “Name:” keyword and continues up to, but not including, the next section’s “Name:” keyword or the end-of-file. Thus the hash includes the blank line(s) at the end of a section and any newline(s) preceding the next “Name:” keyword or end-of-file.

```
//**********************************************************
// Signature Block File Example
//**********************************************************
```

A signature block file is a raw binary file (not base-64 encoded) that is a PKCS#7 defined format signature block. The signature block covers exactly the contents of the signer’s information file. There must be a correspondence between the name of the signer’s information file and the signature block file. The base name matches, and the three-character extension is modified to reflect the signature algorithm used according to the following rules:

- DSA signature algorithm (which uses SHA-1 hash): extension is DSA.
- RSA signature algorithm with MD5 hash: extension is RSA.

So for example with a signer’s information file name of “myinfo.SF,” the corresponding DSA signature block file name would be “myinfo.DSA.”

The format of a signature block file is defined in [PKCS].
This preprocessor symbol gives the value for an attribute inserted in signed manifests to distinguish updates of BIS parameters from updates of other parameters. The representation inserted into the manifest is base-64 encoded.

**Description**

This function updates one of the configurable parameters of the Boot Object Authorization set (Boot Object Authorization Certificate or Boot Authorization Check Flag). It passes back a new unique update token that must be included in the request credential for the next update of any parameter in the Boot Object Authorization set. The token value is unique to this platform, parameter set, and instance of parameter values. In particular, the token changes to a new unique value whenever any parameter in this set is changed.
## Status Codes Returned

<table>
<thead>
<tr>
<th>Status Code</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>EFI_SUCCESS</td>
<td>The function completed successfully.</td>
</tr>
<tr>
<td>EFI_NO_MAPPING</td>
<td>The <code>AppHandle</code> parameter is not or is no longer a valid application</td>
</tr>
<tr>
<td></td>
<td>instance handle associated with the EFI_BIS protocol.</td>
</tr>
<tr>
<td>EFI_OUT_OF_RESOURCES</td>
<td>The function failed due to lack of memory or other resources.</td>
</tr>
<tr>
<td>EFI_DEVICE_ERROR</td>
<td>The function encountered an unexpected internal error in a cryptographic</td>
</tr>
<tr>
<td></td>
<td>software module.</td>
</tr>
</tbody>
</table>
The signed manifest supplied as the `RequestCredential` parameter was invalid (could not be parsed), or
The signed manifest supplied as the `RequestCredential` parameter failed to verify using the installed Boot Object Authorization Certificate or the signer’s Certificate in `RequestCredential`, or
Platform-specific authorization failed, or
The signed manifest supplied as the `RequestCredential` parameter did not include the `X-Intel-BIS-ParameterSet` attribute value, or
The `X-Intel-BIS-ParameterSet` attribute value supplied did not match the required GUID value, or
The signed manifest supplied as the `RequestCredential` parameter did not include the `X-Intel-BIS-ParameterSetToken` attribute value, or
The `X-Intel-BIS-ParameterSetToken` attribute value supplied did not match the platform’s current update-token value, or
The signed manifest supplied as the `RequestCredential` parameter did not include the `X-Intel-BIS-ParameterId` attribute value, or
The `X-Intel-BIS-ParameterId` attribute value supplied did not match one of the permitted values, or
The signed manifest supplied as the `RequestCredential` parameter did not include the `X-Intel-BIS-ParameterValue` attribute value, or
Any other required attribute value was missing, or
The new certificate supplied was too big to store, or
The new certificate supplied was invalid (could not be parsed), or
The new certificate supplied had an unsupported combination of key algorithm and key length, or
The new check flag value supplied is the wrong length (1 byte), or
The signed manifest supplied as the `RequestCredential` parameter did not include a signer certificate, or
The signed manifest supplied as the `RequestCredential` parameter did not include the manifest section named “`memory:UpdateRequestParameters`,” or
The signed manifest supplied as the `RequestCredential` parameter had a signing certificate with an unsupported public-key algorithm, or
The manifest section named “`memory:UpdateRequestParameters`” did not include a digest with a digest algorithm corresponding to the signing certificate’s public key algorithm, or
The zero-length data object referenced by the manifest section named “`memory:UpdateRequestParameters`” did not verify with the digest supplied in that manifest section, or
The signed manifest supplied as the `RequestCredential` parameter did not include a signer’s information file with the `SignerInformationName` identifying attribute value “`BIS_UpdateManifestSignerInfoName`,” or
There were no signers associated with the identified signer’s information file, or
There was more than one signer associated with the identified signer’s information file, or
Any other unspecified security violation occurred.
### EFI_BIS_PROTOCOL.VerifyBootObject()

#### Summary
Verifies the integrity and authorization of the indicated data object according to the indicated credentials.

#### Prototype
```c
typedef EFI_STATUS (EFIAPI *EFI_BIS_VERIFY_BOOT_OBJECT)(
    IN BIS_APPLICATION_HANDLE AppHandle,
    IN EFI_BIS_DATA *Credentials,
    IN EFI_BIS_DATA *DataObject,
    OUT BOOLEAN *IsVerified
);
```

#### Parameters
- **AppHandle**: An opaque handle that identifies the caller’s instance of initialization of the BIS service. Type `BIS_APPLICATION_HANDLE` is defined in the `Initialize()` function description.
- **Credentials**: A Signed Manifest containing verification information for the indicated data object. The Manifest signature itself must meet the requirements described below. This parameter is optional if a Boot Authorization Check is currently not required on this platform (`Credentials.Data` may be `NULL`), otherwise this parameter is required. The required syntax of the Signed Manifest is described in the Related Definition for Manifest Syntax below. Type `EFI_BIS_DATA` is defined in the `Initialize()` function description.
**DataObject**
An in-memory copy of the raw data object to be verified. Type `EFI_BIS_DATA` is defined in the `Initialize()` function description.

**IsVerified**
The function writes `TRUE` if the verification succeeded, otherwise `FALSE`.

**Related Definitions**
```c
//**********************************************************
// Manifest Syntax
//**********************************************************
```

The Signed Manifest consists of three parts grouped together into an Electronic Shrink Wrap archive as described in [SM spec]: a manifest file, a signer’s information file, and a signature block file. These three parts along with examples are described in the following sections. In these examples, text in parentheses is a description of the text that would appear in the signed manifest. Text outside of parentheses must appear exactly as shown. Also note that manifest files and signer’s information files must conform to a 72-byte line-length limit. Continuation lines (lines beginning with a single “space” character) are used for lines longer than 72 bytes. The examples given here follow this rule for continuation lines.

Note that the manifest file and signer’s information file parts of a Signed Manifest are ASCII text files. In cases where these files contain a base-64 encoded string, the string is an ASCII string before base-64 encoding.

```c
//**********************************************************
// Manifest File Example
//**********************************************************
```

The manifest file must include a section referring to a memory-type data object with the reserved name as shown in the example below. This data object is the Boot Object to be verified. An example manifest file is shown below.

```
Manifest-Version: 2.0
ManifestPersistentId: (base-64 representation of a unique GUID)
Name: memory:BootObject
Digest-Algorithms: SHA-1
SHA-1-Digest: (base-64 representation of a SHA-1 digest of the boot object)
```

A line-by-line description of this manifest file is as follows.

- **Manifest-Version: 2.0**
  This is a standard header line that all signed manifests have. It must appear exactly as shown.

- **ManifestPersistentId: (base-64 representation of a unique GUID)**
  The left-hand string must appear exactly as shown. The right-hand string must be a unique GUID for every manifest file created. The Win32 function `UuidCreate()` can be used for this on Win32 systems. The GUID is a binary value that must be base-64 encoded. Base-64 is a simple encoding scheme for representing binary values that uses only printing characters. Base-64 encoding is described in [BASE-64].
**Name: memory:BootObject**

This identifies the section that carries the integrity data for the Boot Object. The string “memory:BootObject” must appear exactly as shown. Note that the Boot Object cannot be found directly from this manifest. A caller verifying the Boot Object integrity must load the Boot Object into memory and specify its memory location explicitly to this verification function through the DataObject parameter.

**Digest-Algorithms: SHA-1**

This enumerates the digest algorithms for which integrity data is included for the data object. For systems with DSA signing, SHA-1 hash, and 1024-bit key length, the digest algorithm must be “SHA-1.” For systems with RSA signing, MD5 hash, and 512-bit key length, the digest algorithm must be “MD5.” Multiple algorithms can be specified as a whitespace-separated list. For every digest algorithm listed, there must also be a corresponding XXX-Digest line.

**SHA-1-Digest: (base-64 representation of a SHA-1 digest of the boot object)**

Gives the corresponding digest value for the data object. The value is base-64 encoded.

```plaintext
//**********************************************************
// Signer’s Information File Example
//**********************************************************
```

The signer’s information file must include a section whose name matches the reserved data object section name of the section in the Manifest file. This section in the signer’s information file carries the integrity data for the corresponding section in the manifest file. An example signer’s information file is shown below.

```plaintext
Signature-Version: 2.0
SignerInformationPersistentId: (base-64 representation of a unique GUID)
SignerInformationName: BIS_VerifiableObjectSignerInfoName

Name: memory:BootObject
Digest-Algorithms: SHA-1
SHA-1-Digest: (base-64 representation of a SHA-1 digest of the corresponding manifest section)
```

A line-by-line description of this signer’s information file is as follows.

**Signature-Version: 2.0**

This is a standard header line that all signed manifests have. It must appear exactly as shown.

**SignerInformationPersistentId: (base-64 representation of a unique GUID)**

The left-hand string must appear exactly as shown. The right-hand string must be a unique GUID for every signer’s information file created. The Win32 function UuidCreate() can be used for this on Win32 systems. The GUID is a binary value that must be base-64 encoded. Base-64 is a simple encoding scheme for representing binary values that uses only printing characters. Base-64 encoding is described in [BASE-64].

**SignerInformationName: BIS_VerifiableObjectSignerInfoName**
The left-hand string must appear exactly as shown. The right-hand string must appear exactly as shown.

Name: memory:BootObject

This identifies the section in the signer's information file corresponding to the section with the same name in the manifest file described earlier. The string "memory:BootObject" must appear exactly as shown.

Digest-Algorithms: SHA-1

This enumerates the digest algorithms for which integrity data is included for the corresponding manifest section. Strings identifying digest algorithms are the same as in the manifest file. The digest algorithms specified here must match those specified in the manifest file. For every digest algorithm XXX listed, there must also be a corresponding XXX-Digest line.

SHA-1-Digest: (base-64 representation of a SHA-1 digest of the corresponding manifest section)

Gives the corresponding digest value for the corresponding manifest section. The value is base-64 encoded. Note that for the purpose of computing the hash of the manifest section, the manifest section starts at the beginning of the opening “Name:” keyword and continues up to, but not including, the next section's “Name:” keyword or the end-of-file. Thus the hash includes the blank line(s) at the end of a section and any newline(s) preceding the next “Name:” keyword or end-of-file.

//**********************************************************
// Signature Block File Example
//**********************************************************

A signature block file is a raw binary file (not base-64 encoded) that is a PKCS#7 defined format signature block. The signature block covers exactly the contents of the signer's information file. There must be a correspondence between the name of the signer's information file and the signature block file. The base name matches, and the three-character extension is modified to reflect the signature algorithm used according to the following rules:

- DSA signature algorithm (which uses SHA-1 hash): extension is DSA.
- RSA signature algorithm with MD5 hash: extension is RSA.

So for example with a signer's information file name of “myinfo.SF,” the corresponding DSA signature block file name would be “myinfo.DSA.”

The format of a signature block file is defined in [PKCS].

Description

This function verifies the integrity and authorization of the indicated data object according to the indicated credentials. The rules for successful verification depend on whether or not a Boot Authorization Check is currently required on this platform.

If a Boot Authorization Check is not currently required on this platform, no authorization check is performed. However, the following rules are applied for an integrity check:

- In this case, the credentials are optional. If they are not supplied (Credentials.Data is NULL), no integrity check is performed, and the function returns immediately with a “success” indication and IsVerified is TRUE.
If the credentials are supplied (Credentials.Data is other than NULL), integrity checks are performed as follows:

- Verify the credentials – The credentials parameter is a valid signed Manifest, with a single signer. The signer’s identity is included in the credential as a certificate.
- Verify the data object – The Manifest must contain a section named “memory:BootObject,” with associated verification information (in other words, hash value). The hash value from this Manifest section must match the hash value computed over the specified DataObject data.
- If these checks succeed, the function returns with a “success” indication and IsVerified is TRUE. Otherwise, IsVerified is FALSE and the function returns with a “security violation” indication.

If a Boot Authorization Check is currently required on this platform, authorization and integrity checks are performed. The integrity check is the same as in the case above, except that it is required. The following rules are applied:

- Verify the credentials – The credentials parameter is required in this case (Credentials.Data must be other than NULL). The credentials parameter is a valid Signed Manifest, with a single signer. The signer’s identity is included in the credential as a certificate.
- Verify the data object – The Manifest must contain a section named “memory:BootObject,” with associated verification information (in other words, hash value). The hash value from this Manifest section must match the hash value computed over the specified DataObject data.
- Do Authorization check – This happens one of two ways depending on whether or not the platform currently has a Boot Object Authorization Certificate configured.
  - If a Boot Object Authorization Certificate is not currently configured, this function interacts with the user in a platform-specific way to determine whether the operation should succeed.
  - If a Boot Object Authorization Certificate is currently configured, this function uses the Boot Object Authorization Certificate to determine whether the operation should succeed. The public key certified by the signer’s certificate must match the public key in the Boot Object Authorization Certificate configured for this platform. The match must be direct, that is, the signature authority cannot be delegated along a certificate chain.
- If these checks succeed, the function returns with a “success” indication and IsVerified is TRUE. Otherwise, IsVerified is FALSE and the function returns with a “security violation” indication.

Note that if a Boot Authorization Check is currently required on this platform this function always performs an authorization check, either through platform-specific user interaction or through a signature generated with the private key corresponding to the public key in the platform’s Boot Object Authorization Certificate.
### Status Codes Returned

<table>
<thead>
<tr>
<th>Code</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>EFI_SUCCESS</td>
<td>The function completed successfully.</td>
</tr>
<tr>
<td>EFI_NO_MAPPING</td>
<td>The <code>AppHandle</code> parameter is not or is no longer a valid application instance handle associated with the EFI_BIS protocol.</td>
</tr>
<tr>
<td>EFI_INVALID_PARAMETER</td>
<td>The <code>Credentials</code> parameter supplied by the caller is <code>NULL</code> or an invalid memory reference, or</td>
</tr>
<tr>
<td></td>
<td>The Boot Authorization Check is currently required on this platform and the <code>Credentials.Data</code> parameter supplied by the caller is <code>NULL</code> or an invalid memory reference, or</td>
</tr>
<tr>
<td></td>
<td>The <code>DataObject</code> parameter supplied by the caller is <code>NULL</code> or an invalid memory reference, or</td>
</tr>
<tr>
<td></td>
<td>The <code>DataObject.Data</code> parameter supplied by the caller is <code>NULL</code> or an invalid memory reference, or</td>
</tr>
<tr>
<td></td>
<td>The <code>IsVerified</code> parameter supplied by the caller is <code>NULL</code> or an invalid memory reference.</td>
</tr>
<tr>
<td>EFI_OUT_OF_RESOURCES</td>
<td>The function failed due to lack of memory or other resources.</td>
</tr>
<tr>
<td>EFI_SECURITY_VIOLATION</td>
<td>The signed manifest supplied as the <code>Credentials</code> parameter was invalid (could not be parsed), or</td>
</tr>
<tr>
<td></td>
<td>The signed manifest supplied as the <code>Credentials</code> parameter failed to verify using the installed Boot Object Authorization Certificate or the signer’s Certificate in <code>Credentials</code>, or</td>
</tr>
<tr>
<td></td>
<td>Platform-specific authorization failed, or</td>
</tr>
<tr>
<td></td>
<td>Any other required attribute value was missing, or</td>
</tr>
<tr>
<td></td>
<td>The signed manifest supplied as the <code>Credentials</code> parameter did not include a signer certificate, or</td>
</tr>
</tbody>
</table>
EFI_BIS_PROTOCOL.VerifyObjectWithCredential()

Summary
Verifies the integrity and authorization of the indicated data object according to the indicated credentials and authority certificate.
Prototype

typedef
EFI_STATUS
(EFIAPI *EFI_BIS_VERIFY_OBJECT_WITH_CREDENTIAL)(
    IN BIS_APPLICATION_HANDLE AppHandle,
    IN EFI_BIS_DATA *Credentials,
    IN EFI_BIS_DATA *DataObject,
    IN EFI_BIS_DATA *SectionName,
    IN EFI_BIS_DATA *AuthorityCertificate,
    OUT BOOLEAN *IsVerified
);

Parameters

AppHandle An opaque handle that identifies the caller’s instance of initialization of the BIS service. Type BIS_APPLICATION_HANDLE is defined in the Initialize() function description.

Credentials A Signed Manifest containing verification information for the indicated data object. The Manifest signature itself must meet the requirements described below. The required syntax of the Signed Manifest is described in the Related Definition of Manifest Syntax below. Type EFI_BIS_DATA is defined in the Initialize() function description.

DataObject An in-memory copy of the raw data object to be verified. Type EFI_BIS_DATA is defined in the Initialize() function description.

SectionName An ASCII string giving the section name in the manifest holding the verification information (in other words, hash value) that corresponds to DataObject. Type EFI_BIS_DATA is defined in the Initialize() function description.

AuthorityCertificate A digital certificate whose public key must match the signer’s public key which is found in the credentials. This parameter is optional (AuthorityCertificate.Data may be NULL). Type EFI_BIS_DATA is defined in the Initialize() function description.

IsVerified The function writes TRUE if the verification was successful. Otherwise, the function writes FALSE.

Related Definitions

//**********************************************************
// Manifest Syntax
//**********************************************************

The Signed Manifest consists of three parts grouped together into an Electronic Shrink Wrap archive as described in [SM spec]: a manifest file, a signer’s information file, and a signature block file. These three parts along with examples are described in the following sections. In these examples, text in parentheses
is a description of the text that would appear in the signed manifest. Text outside of parentheses must appear exactly as shown. Also note that manifest files and signer’s information files must conform to a 72-byte line-length limit. Continuation lines (lines beginning with a single “space” character) are used for lines longer than 72 bytes. The examples given here follow this rule for continuation lines.

Note that the manifest file and signer’s information file parts of a Signed Manifest are ASCII text files. In cases where these files contain a base-64 encoded string, the string is an ASCII string before base-64 encoding.

//**********************************************************
// Manifest File Example
//**********************************************************

The manifest file must include a section referring to a memory-type data object with the caller-chosen name as shown in the example below. This data object is the Data Object to be verified. An example manifest file is shown below.

**Manifest-Version**: 2.0  
**ManifestPersistentId**: (base-64 representation of a unique GUID)

**Name**: (a memory-type data object name)  
**Digest-Algorithms**: SHA-1  
**SHA-1-Digest**: (base-64 representation of a SHA-1 digest of the data object)

A line-by-line description of this manifest file is as follows.

**Manifest-Version**: 2.0  
This is a standard header line that all signed manifests have. It must appear exactly as shown.

**ManifestPersistentId**: (base-64 representation of a unique GUID)

The left-hand string must appear exactly as shown. The right-hand string must be a unique GUID for every manifest file created. The Win32 function UuidCreate() can be used for this on Win32 systems. The GUID is a binary value that must be base-64 encoded. Base-64 is a simple encoding scheme for representing binary values that uses only printing characters. Base-64 encoding is described in [BASE-64].

**Name**: (a memory-type data object name)

This identifies the section that carries the integrity data for the target Data Object. The right-hand string must obey the syntax for memory-type references, that is, it is of the form “memory:SomeUniqueName.” The “memory:” part of this string must appear exactly. The “SomeUniqueName” part is chosen by the caller. It must be unique within the section names in this manifest file. The entire “memory:SomeUniqueName” string must match exactly the corresponding string in the signer’s information file described below. Furthermore, this entire string must match the value given for the SectionName parameter to this function. Note that the target Data Object cannot be found directly from this manifest. A caller verifying the Data Object integrity must load the Data Object into memory and specify its memory location explicitly to this verification function through the DataObject parameter.

**Digest-Algorithms**: SHA-1
This enumerates the digest algorithms for which integrity data is included for the data object. For systems with DSA signing, SHA-1 hash, and 1024-bit key length, the digest algorithm must be “SHA-1.” For systems with RSA signing, MD5 hash, and 512-bit key length, the digest algorithm must be “MD5.” Multiple algorithms can be specified as a whitespace-separated list. For every digest algorithm listed, there must also be a corresponding `XXX-Digest` line.

**SHA-1-Digest:** (base-64 representation of a SHA-1 digest of the data object)

Gives the corresponding digest value for the data object. The value is base-64 encoded.

```plaintext
//**********************************************************
// Signer's Information File Example
//**********************************************************
```

The signer’s information file must include a section whose name matches the reserved data object section name of the section in the Manifest file. This section in the signer’s information file carries the integrity data for the corresponding section in the manifest file. An example signer’s information file is shown below.

```plaintext
Signature-Version: 2.0
SignerInformationPersistentId: (base-64 representation of a unique GUID)
SignerInformationName: BIS_VerifiableObjectSignerInfoName
Name: (a memory-type data object name)
Digest-Algorithms: SHA-1
SHA-1-Digest: (base-64 representation of a SHA-1 digest of the corresponding manifest section)
```

A line-by-line description of this signer’s information file is as follows.

**Signature-Version: 2.0**

This is a standard header line that all signed manifests have. It must appear exactly as shown.

**SignerInformationPersistentId:** (base-64 representation of a unique GUID)

The left-hand string must appear exactly as shown. The right-hand string must be a unique GUID for every signer’s information file created. The Win32 function `UuidCreate()` can be used for this on Win32 systems. The GUID is a binary value that must be base-64 encoded. Base-64 is a simple encoding scheme for representing binary values that uses only printing characters. Base-64 encoding is described in [BASE-64].

**SignerInformationName:** BIS_VerifiableObjectSignerInfoName

The left-hand string must appear exactly as shown. The right-hand string must appear exactly as shown.

**Name:** (a memory-type data object name)

This identifies the section in the signer’s information file corresponding to the section with the same name in the manifest file described earlier. The right-hand string must match exactly the corresponding string in the manifest file described above.

**Digest-Algorithms:** SHA-1
This enumerates the digest algorithms for which integrity data is included for the corresponding manifest section. Strings identifying digest algorithms are the same as in the manifest file. The digest algorithms specified here must match those specified in the manifest file. For every digest algorithm XXX listed, there must also be a corresponding XXX-Digest line.

SHA-1-Digest: (base-64 representation of a SHA-1 digest of the corresponding manifest section)

Gives the corresponding digest value for the corresponding manifest section. The value is base-64 encoded. Note that for the purpose of computing the hash of the manifest section, the manifest section starts at the beginning of the opening “Name:” keyword and continues up to, but not including, the next section’s “Name:” keyword or the end-of-file. Thus the hash includes the blank line(s) at the end of a section and any newline(s) preceding the next “Name:” keyword or end-of-file.

//**********************************************************
// Signature Block File Example
//**********************************************************

A signature block file is a raw binary file (not base-64 encoded) that is a PKCS#7 defined format signature block. The signature block covers exactly the contents of the signer’s information file. There must be a correspondence between the name of the signer’s information file and the signature block file. The base name matches, and the three-character extension is modified to reflect the signature algorithm used according to the following rules:

- DSA signature algorithm (which uses SHA-1 hash): extension is DSA.
- RSA signature algorithm with MD5 hash: extension is RSA.

So for example with a signer’s information file name of “myinfo.SF,” the corresponding DSA signature block file name would be “myinfo.DSA.”

The format of a signature block file is defined in [PKCS].

Description

This function verifies the integrity and authorization of the indicated data object according to the indicated credentials and authority certificate.

Both an integrity check and an authorization check are performed. The rules for a successful integrity check are:

- Verify the credentials – The credentials parameter is a valid Signed Manifest, with a single signer. The signer’s identity is included in the credential as a certificate.
- Verify the data object – The Manifest must contain a section with the name as specified by the SectionName parameter, with associated verification information (in other words, hash value). The hash value from this Manifest section must match the hash value computed over the data specified by the DataObject parameter of this function.

The authorization check is optional. It is performed only if the AuthorityCertificate.Data parameter is other than NULL. If it is other than NULL, the rules for a successful authorization check are:

- The AuthorityCertificate parameter is a valid digital certificate. There is no requirement regarding the signer (issuer) of this certificate.
- The public key certified by the signer’s certificate must match the public key in the AuthorityCertificate. The match must be direct, that is, the signature authority cannot be delegated along a certificate chain.

If all of the integrity and authorization check rules are met, the function returns with a “success” indication and IsVerified is TRUE. Otherwise, it returns with a nonzero specific error code and IsVerified is FALSE.

### Status Codes Returned

<table>
<thead>
<tr>
<th>Status Code</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>EFI_SUCCESS</td>
<td>The function completed successfully.</td>
</tr>
<tr>
<td>EFI_NO_MAPPING</td>
<td>The AppHandle parameter is not or is no longer a valid application instance handle associated with the EFI_BIS protocol.</td>
</tr>
<tr>
<td>EFI_INVALID_PARAMETER</td>
<td>The Credentials parameter supplied by the caller is NULL or an invalid memory reference, or The Credentials.Data parameter supplied by the caller is NULL or an invalid memory reference, or The Credentials.Length supplied by the caller is zero, or The DataObject parameter supplied by the caller is NULL or an invalid memory reference, or The DataObject.Data parameter supplied by the caller is NULL or an invalid memory reference, or</td>
</tr>
<tr>
<td>EFI_INVALID_PARAMETER</td>
<td>The SectionName parameter supplied by the caller is NULL or an invalid memory reference, or The SectionName.Data parameter supplied by the caller is NULL or an invalid memory reference, or The SectionName.Length supplied by the caller is zero, or The AuthorityCertificate parameter supplied by the caller is NULL or an invalid memory reference, or The IsVerified parameter supplied by the caller is NULL or an invalid memory reference.</td>
</tr>
<tr>
<td>EFI_OUT_OF_RESOURCES</td>
<td>The function failed due to lack of memory or other resources.</td>
</tr>
</tbody>
</table>
EFI_SECURITY_VIOLATION

<table>
<thead>
<tr>
<th>EFI_SECURITY_VIOLATION</th>
</tr>
</thead>
<tbody>
<tr>
<td>The Credentials.Data supplied by the caller is NULL, or</td>
</tr>
<tr>
<td>The AuthorityCertificate supplied by the caller was invalid (could not be parsed), or</td>
</tr>
<tr>
<td>The signed manifest supplied as Credentials failed to verify using the AuthorityCertificate supplied by the caller or the manifest’s signer’s certificate, or</td>
</tr>
<tr>
<td>Any other required attribute value was missing, or</td>
</tr>
<tr>
<td>The signed manifest supplied as the Credentials parameter did not include a signer certificate, or</td>
</tr>
<tr>
<td>The signed manifest supplied as the Credentials parameter did not include the manifest section named according to SectionName, or</td>
</tr>
<tr>
<td>The signed manifest supplied as the Credentials parameter had a signing certificate with an unsupported public-key algorithm, or</td>
</tr>
<tr>
<td>The manifest section named according to SectionName did not include a digest with a digest algorithm corresponding to the signing certificate’s public key algorithm, or</td>
</tr>
<tr>
<td>The data object supplied as the DataObject parameter and referenced by the manifest section named according to SectionName did not verify with the digest supplied in that manifest section, or</td>
</tr>
<tr>
<td>EFI_SECURITY_VIOLATION The signed manifest supplied as the Credentials parameter did not include a signer’s information file with the SignerInformationName identifying attribute value “BIS_VerifiableObjectSignerInfoName,” or</td>
</tr>
<tr>
<td>There were no signers associated with the identified signer’s information file, or</td>
</tr>
<tr>
<td>There was more than one signer associated with the identified signer’s information file, or</td>
</tr>
<tr>
<td>Any other unspecified security violation occurred.</td>
</tr>
</tbody>
</table>

EFI_DEVICE_ERROR

<table>
<thead>
<tr>
<th>EFI_DEVICE_ERROR</th>
</tr>
</thead>
<tbody>
<tr>
<td>An unexpected internal error occurred while attempting to retrieve the public key algorithm of the manifest’s signer’s certificate, or</td>
</tr>
<tr>
<td>An unexpected internal error occurred in a cryptographic software module.</td>
</tr>
</tbody>
</table>

24.6 DHCP options for ISCSI on IPV6

Option 59 is the iSCSI Root path
The format of the root path is

"iscsi:"<servername>":"<protocol>":"<port>":"<LUN>":"<targetname>

This is per the description in IETF RFC 4173. See https://uefi.org/uefi#RFC4173 for a link to this document.

Option 60 is the DHCP Server address.

This is formatted the same as parameter 1 in OPT_BOOTFILE_PARAM (60) of the IPv6 address of the DHCP server (IETF RFC 5970). See https://uefi.org/uefi#RFC5970 for a link to this document.

24.7 HTTP Boot

24.7.1 Boot from URL

Elsewhere in this specification there is defined a discoverable network boot using DHCP as a control channel allowing a firmware client machine export its architecture type, and then have the boot server response with a binary image. For the UEFI architecture types defined in “Links to UEFI-Related Documents” (http://uefi.org/uefi) under the heading “IANA DHCPv6 parameters”, the binary image on the boot service is a UEFI-formatted executable with a machine subsystem type that corresponds to the UEFI firmware on the client machine, or it could be mounted as a RAM disk which contains a UEFI-compliant file system (see Section 13.3). This binary image is often referred to as a “Network Boot Program” (NBP). The UEFI client machine that downloads the NBP uses the IPV4 or IPV6 TFTP protocol to address the indicated server, depending upon if DHCP4 or DHCP6 was used initially, in order to download images such as 64-bit UEFI (type 0x07).

This section defines a related method indicated by other codes in the DHCP options, in which the name and path of the NBP are specified as a URI string in one of several formats specifying protocol and unique name identifying the NBP for the specified protocol. In this method the NBP will be downloaded via IPV4 or IPV6 HTTP protocol if the tag indicates x64 UEFI HTTP Boot (type code 0x0f for x86 and 0x10 for x64).

In the future other protocols such as FTP or NFS could be encoded with both new tag types and corresponding URIs (e.g., ‘ftp:///nbp.efi or nfs:///nbp.efi, respectively). However, assignment of these type codes has not yet occurred.

The rest of this section will describe ‘HTTP Boot’ as one example of ‘boot from URI’. It is expected that the procedure can be extended as additional protocol type codes are defined.

Please reference the definitions of EFI_DNS4_PROTOCOL and EFI_DNS6_PROTOCOL elsewhere in this document. In systems that also support one of both of these protocols, the target URI can be specified using Internet domain name format understood by DNS servers supporting the appropriate RFC specifications.

Also, elsewhere in this document, the PXE2.1 and UEFI2.4 netboot6 sections talk about the ‘boot from TFTP’ method of ‘boot from URI.’

The following RFC documents should be consulted for network message details related to the processes described in this chapter:

1. RFC1034 - "Domain Names - Concepts and Facilities",
2. RFC 1035 - "Domain Names - Implementation and Specification",
24.7.2 Concept configuration for a typical HTTP Boot scenario

HTTP Boot is client-server communication based application. It combines the DHCP, DNS, and HTTP protocols to provide the capability for system deployment and configuration over the network. This new capability can be utilized as a higher-performance replacement for tftp-based PXE boot methods of network deployment.

24.7.2.1 Use in Corporate environment

A typical network configuration which supports UEFI HTTP Boot may involve one or more UEFI client systems, and several server systems. Figure 24-4 show a typical HTTP Boot network topology for a corporate environment.

- **UEFI HTTP Boot Client** initiates the communication between the client and different server system.
- **DHCP server with HTTPBoot extension** for boot service discovery. Besides the standard host configuration information (such as address/subnet/gateway/name-server, etc...), the DHCP
server with the extensions can also provide the discovery of URI locations for boot images on the HTTP server.

- **HTTP server** could be located either inside the corporate environment or across networks, such as on the Internet. The boot resource itself is deployed on the HTTP server. In this example, “http://webserver/boot/boot.efi” is used as the boot resource. Such an application is also called a Network Boot Program (NBP). NBPs are used to setup the client system, which may include installation of an operating system, or running a service OS for maintenance and recovery tasks.

- **DNS server** is optional; and provides standard domain name resolution service.

### 24.7.2.2 Use case in Home environment

In a corporate environment, a standard DHCP server can be enhanced to support the HTTPBoot extension. In a home network, generally only an optional standard DHCP server may be available for host configuration information assignment. Figure 24-5 shows the concept network topology for a typical home PC environment.

![Figure 24-5 HTTP Boot Network Topology Concept2 – Home environments](image)

**UEFI HTTP Boot Client** initiates the communication between the client and different servers. In the home configuration however, the client will expect the boot resource information to be available from a source other than the standard DHCP server, and that source does not typically have HTTPBoot extensions. Instead of DHCP, the boot URI could be created by a UEFI application or extracted from text entered by a user.

**DHCP server** is optional, and if available in the network, provides the standard service to assign host configuration information to the UEFI Client (e.g. address/subnet/gateway/name-server/etc.). In case the standard DHCP server is not available, the same host configuration information should be provided by a UEFI application or extracted from text entered by a user prior to the client initiating the communication.

**DNS Server** is optional, and provides standard domain name resolution service.
24.7.3 Protocol Layout for UEFI HTTP Boot Client concept configuration for a typical HTTP Boot scenario

This figure illustrates the UEFI network layers related to how the HTTP Boot works.

The HTTP Boot driver is layered on top of a UEFI Network stack implementation. It consumes DHCP service to do the Boot service discovery, and DNS service to do domain name resolution if needed. It also consumes HTTP serviced to retrieve images from the HTTP server. The functionality needed in the HTTP Boot scenario is limited to client initiated requests to download the boot image.

TLS is consumed if HTTPS functionality is needed. The TLS design is covered in Section 28.10.2.

The HTTP Boot driver produces **LoadFile** protocol and device path protocol. BDS will provide the boot options for the HTTP Boot. Once a boot option for HTTP boot is executed, a particular network interface is selected. HTTP Boot driver will perform all steps on that interface and it is not required to use other interfaces.
24.7.3.1 Device Path

If both IPv4 and IPv6 are supported, the HTTP Boot driver should create two child handles, with LoadFile and DevicePath installed on each child handle. For the device path, an IP device path node and a BootURI device path are appended to the parent device path, for example:

PciRoot(0x0)/Pci(0x19, 0x0)/MAC(001230F4B4FF, 0x0)/IPv4(0.0.0.0, 0, DHCP, 0.0.0.0, 0.0.0.0, 0.0.0.0)/Uri()
PciRoot(0x0)/Pci(0x19, 0x0)/MAC(001230F4B4FF, 0x0)/IPv6(::/128, 0, Static, ::/128, ::/128, 0)/Uri()

Also, after retrieving the boot resource information and IP address, the BootURI device path node will be updated to include the BootURI information. For example, if the NBP is a UEFI-formatted executable, the device patch will be updated to

PciRoot(0x0)/Pci(0x19, 0x0)/MAC(001230F4B4FF, 0x0)/IPv4(192.168.1.100, TCP, DHCP, 192.168.1.5, 192.168.1.1, 255.255.255.0)/Uri(http://192.168.1.100/shell.efi)
PciRoot(0x0)/Pci(0x19, 0x0)/MAC(001230F4B4FF, 0x0)/IPv6(2015::100, TCP, StatefulAutoConfigure, 2015::5, 2015::10, 64)/Uri(http://2015::100/shell.efi)

These two instances allow for the boot manager to decide a preference of IPv6 versus IPv4.

If the NBP is a binary image which could be mounted as a RAM disk, the device path will be updated to

PciRoot(0x0)/Pci(0x19, 0x0)/MAC(001230F4B4FF, 0x0)/IPv4(192.168.1.100, TCP, DHCP, 192.168.1.5, 192.168.1.1, 255.255.255.0)/Uri(http://192.168.1.100/boot.iso [%])
PciRoot(0x0)/Pci(0x19, 0x0)/MAC(001230F4B4FF, 0x0)/IPv6(2015::100, TCP, StatefulAutoConfigure, 2015::5, 2015::10, 64)/Uri(http://2015::100/boot.iso)

In this case, the HTTP Boot driver will register RAM disk with the downloaded NBP, by appending a RamDisk device node to the device path above, like

PciRoot(0x0)/Pci(0x19, 0x0)/MAC(001230F4B4FF, 0x0)/IPv4(192.168.1.100, TCP, DHCP, 192.168.1.5, 192.168.1.1, 255.255.255.0)/Uri(http://192.168.1.100/boot.iso)/RamDisk(0x049EA000, 0x5DEA000, 0, 3D5ABD30-4175-87CE-6D64-D2ADE523C4BB)
PciRoot(0x0)/Pci(0x19, 0x0)/MAC(001230F4B4FF, 0x0)/IPv6(2015::100, TCP, StatefulAutoConfigure, 2015::5, 2015::10, 64)/Uri(http://2015::100/boot.iso)/RamDisk(0x049EA000, 0x5DEA000, 0, 3D5ABD30-4175-87CE-6D64-D2ADE523C4BB)
In some cases, Uri includes a host name and DNS become mandatory for translating the name to the IP address of the host. The HTTP Boot driver may append DNS device path node immediately before Uri device path node, for example:

```
PciRoot(0x0)/Pci(0x19, 0x0)/MAC(001230F4B4FF, 0x0)/IPv4(192.168.1.100, TCP, DHCP, 192.168.1.5, 192.168.1.1, 255.255.255.0)/Dns(192.168.22.100, 192.168.22.101)/Uri(http://www.bootserver.com/boot.iso)/RamDisk(0x049EA000, 0x5DEA000, 0, 3D5ABD30-4175-87CE-6D64-D2ADE523C4BB)
PciRoot(0x0)/Pci(0x19, 0x0)/MAC(001230F4B4FF, 0x0)/IPv6(2015::100, TCP, StatefulAutoConfigure, 2015::5, 2015::10, 64)/Dns(2016::100, 2016::101)/Uri(http://www.bootserver.com/boot.iso)/ RamDisk(0x049EA000, 0x5DEA000, 0, 3D5ABD30-4175-87CE-6D64-D2ADE523C4BB)
```

If HTTP Boot driver cannot obtain the DNS server addresses, it should not append an empty DNS device path node.

The boot manager could use the example device paths to match the device which produces a device path protocol including a URI device path node in the system, without matching the Specific Device Path data in IP device path node and URI device path node, because the IP device path node and URI device path node might be updated by HTTP Boot driver in different scenarios.

The BootURI information could be retrieved from a DHCP server with HTTPBoot extension, or from a boot option which includes a short-form URI device path, or from a boot option which includes a URI device path node, or created by a UEFI application or extracted from text entered by a user.

Once the HTTP Boot driver retrieves the BootURI information from the short-form URI device path, it will perform all other steps for HTTP boot except retrieving the BootURI from DHCP server. Also, when the short-form URI device path is inputted to HTTP Boot driver via LoadFile protocol, the HTTP Boot driver should expand the short-form URI device path to above example device path after retrieving IP address configuration (address, subnet, gateway, and optionally the name-server) from the DHCP server. In case of the home environment with no DHCP server, the same information may be provisioned by OEM or input by the end user through Setup Options. The IP and optional DNS device path nodes, constructed with this information and prefixed to the short-form URI device path, can be inputted to the HTTP Boot driver via LoadFile protocol. The name server information in the form of DNS device path node is optional, and is used only when the BootURI contains the server name or FQDN. The HTTP Boot driver will then consume the information in the device path and initiate the necessary communication.

Once the HTTP Boot driver retrieves the BootURI information from a boot option which includes a URI device path node, it should retrieve the IP address configuration from the IP device path node of the same boot option. If the IP address configuration or BootURI information is empty, the HTTP Boot driver could retrieve the required information from DHCP server. If the IP address configuration or BootURI information is not empty but invalid, the HTTP boot process will fail.

The HTTP Boot block diagram (Figure 24-6) describes a suggested implementation for HTTP Boot. Other implementation can create their own HTTP Netboot Driver which meets the requirements for their netboot methodology.

### 24.7.4 Concept of Message Exchange in a typical HTTP Boot scenario (IPv4 in Corporate Environment)

In summary, the newly installed networked client machine (UEFI HTTP Boot Client) should be able to enter a heterogeneous network, acquire a network address from a DHCP server, and then download an NBP to set itself up.
The concept of HTTP Boot message exchange sequence is as follows. The client initiates the DHCPv4 D.O.R.A process by broadcasting a DHCPDISCOVER containing the extension that identifies the request as coming from a client that implements the HTTP Boot functionality. Assuming that a DHCP server or a Proxy DHCP server implementing this extension is available, after several intermediate steps, besides the standard configuration such as address/subnet/router/dns-server, boot resource location will be provided to the client system in the format of a URI. The URI points to the NBP which is appropriate for this client hardware configuration. A boot option is created, and if selected by the system logic the client then uses HTTP to download the NBP from the HTTP server into memory. Finally, the client executes the downloaded NBP image from memory. This image can then consume other UEFI interfaces for further system setup.
24.7.4.1 Message exchange between EFI Client and DHCP server using DHCP Client Extensions

24.7.4.1.1 Client broadcast
The client broadcasts a DHCP Discover message to the standard DHCP port (67).
An option field in this packet contains the following:

- Fill DHCP option 55 – Parameter Requested List option
  - Address configuration, Server information, Name server, Vendor class identifier
- A DHCP option 97: UUID/GUID-based Client Identifier
- A DHCP option 94: Client Network Identifier Option
  - If support UNDI, fill this option (Refer RFC5970)
- A DHCP option 93: the client system architecture (Refer [Arch-Type])
  - 0x0F - x86 UEFI HTTP Boot
  - 0x10 - x64 UEFI HTTP Boot
- A DHCP option 60, Vendor Class ID, set to “HTTPClient:Arch:XXXX:UNDI:YYYYZZZ”

24.7.4.1.2 DHCP server response
The DHCP server responds by sending DHCPOFFER message on standard DHCP reply port (68).
The HTTP Boot Client may possibly receive multiple DHCPOFFER packets from different sources of DHCP Services, possibly from DHCP Services which recognize the HTTP extensions or from Standard DHCP Services.

A service recognizing HTTP extensions must respond with an offer that has Option 60 (Vendor class identifier) parameter set to “HTTPClient”, in response to the Vendor class identifier requested in option 55 in the DHCP Discover message.

Each message contains standard DHCP parameters: an IP address for the client and any other parameters that the administrator might have configured on the DHCP or Proxy DHCP Service. The DHCP service or Proxy DHCP which recognizes the HTTPBoot extension will provide DHCPOFFER with HTTPClient extensions. If this is a Proxy DHCP service, then the client IP address field is (0.0.0.0). If this is from a DHCP service, then the returned client IP address field is valid.

From the received DHCPOFFER(s), the client records the information as follows:

- Client IP address (and other parameters) offered by a standard DHCP/BOOTP services.
- If Boot URI information is provided thru ‘file’ field in DHCP Header or option 67, then the client will record the information as well.
- Optional Name-server information if URI is displayed using domain-name

**Timeout**: After Client sent out the DHCP Discover packet, the Client will wait for a timeout to collect enough DHCP Offers. If failed to retrieve all the required information, the DHCP Discover will be retried, at most four times. The four timeout mechanisms is 4, 8, 16 and 32 seconds respectively,

**Priority**: Among all the received DHCPOFFERs, the Priority is considered as follows:
Priority 1
Choose the DHCPOFFER that provides all required information:

<IP address configuration, Boot URI configuration, Name-server configuration (if domain-name used in Boot URI)>

If Boot URI and IP address configuration provided in different DHCPOFFER, Using 5 DHCPOFFER as example for priority description

- Packet1 – DHCPOFFER, provide <IP address configuration, Name server>
- Packet2 – DHCPOFFER, provide <IP address configuration>
- Packet3 – DHCPOFFER, provide <domain-name expressed URI>
- Packet4 – DHCPOFFER, provide <IP address expressed URI>
- Packet5 – DHCPOFFER, provide <IP address, domain-name expressed URI>

Then,

Priority 2
Choose the DHCPOFFER from different packet, firstly find out URI info represented in IP address mode, then choose DHCPOFFER which provide IP address configuration

In this example, the chosen DHCPOFFER packet is packet4 + packet1 / packet2 (packet 1/2 take same priority, implementation can make its own implementation choice)

Priority 3
Choose the DHCPOFFER from different packet, firstly find out URI info represented in domain-name mode, then choose DHCPOFFER which provide <IP address configuration, domain-name expressed URI>

In this example, the chosen DHCPOFFER packet is packet3 / packet5 + packet1

Note: If packet5, then client IP address assigned by Packet5 will be override by IP address in packet1.

Priority 4
If failed to retrieve <Boot URI / IP address / (on-demand) Name-server> information through all received DHCPOFFERs, this is deemed as FAILED-CONFIGURATION

Assuming the boot image is in the boot subdirectory of the web server root, the supported URI could be one of below formats. [RFC3986] where ‘/boot/’ is replaced by administrator-created directory, and ‘image’ is the file name of the NBP.

http://reg-name:port/boot/image
http://ipv4address:port/boot/image
http://ipv6address:port/boot/image

In the URL example, Port is optional if web service is provided through port 80 for the HTTP data transfer. Commonly, the reg-name use DNS as name registry mechanism.

After retrieving the boot URI through Section 24.7.3.1, if IP address (either IPv4 or IPv6 address) is provided, the HTTP Boot Client can directly use that address for next step HTTP transfer. If a reg-name is
provided in the URI, the HTTP Boot Client driver need initiate DNS process (Section 24.7.4.3) to resolve its IP address.

24.7.4.1.3 DHCP Request

The HTTP Boot Client selects an IP address offered by a DHCP Service, and then it completes the standard DHCP protocol by sending a DHCP Request packet for the address to the DHCP Server and waiting for acknowledgement from the DHCP server.

24.7.4.1.4 DHCP ACK

The server acknowledges the IP address by sending DHCP ACK packet to the client.

24.7.4.2 Message exchange between UEFI Client and DHCP server not using DHCP Client Extensions

In a home environment, because the Boot URI Information will not be provided by the DHCP Offers, we need other channels to provide this information. The implementation suggestion is provisioning this information by OEM or input by end user through Setup Options, henceforth, the UEFI Boot Client already know the Boot URI before contacting the DHCP server.

The message exchange between the EFI Client and DHCP server will be standard DHCP D.O.R.A to obtain <IP address, Name-server>.

In the case of a home environment without a DHCP server, the above message exchange is not needed, and the UEFI HTTP Boot Client will have the <IP address, Name-server> provisioned by OEM or input by the end user through Setup Options.

24.7.4.3 Message in DNS Query/Reply

The DNS Query/Reply is a standard process defined in DNS Protocol [RFC 1034, RFC 1035]. Multiple IP address might be retrieved from the DNS process. It’s the HTTP Boot Client driver’s responsibility to select proper IP address automatically or expose user interface for customer to decide proper IP address.

24.7.4.4 Message in HTTP Download

In the HTTP Boot scenario, HTTP GET message is used to get image from the Web server.

24.7.5 Concept of Message Exchange in HTTP Boot scenario (IPv6)

24.7.5.1 Message exchange between EFI Client and DHCPv6 server with DHCP Client extensions

24.7.5.1.1 Client multicast a DHCPv6 Solicit message to the standard DHCPv6 port (547).

Besides the options required for address auto-configuration, option field in this packet also contains the following:

- Fill DHCPv6 Option 6 – Option Request Option
  - Request server to supply option 59 (OPT_BOOTFILE_URL), option 60 (OPT_BOOTFILE_PARAM), option 23 (OPT_DNS_SERVERS), option 16 (OPTION_VENDOR_CLASS).
• A DHCPv6 option 1, Client identifier
• A DHCPv6 option 16, Vendor Class ID, set to “HTTPClient:Arch:XXXX:UNDI:YYYYZZZ”
• A DHCPv6 option 61: the client system architecture (Refer [Arch-Type])
  — 0x0F - x86 UEFI HTTP Boot
  — 0x10 - x64 UEFI HTTP Boot
• A DHCPv6 option 62: Client Network Identifier Option
  — If support UNDI, fill this option (Refer RFC5970)

24.7.5.1.2 Server unicast DHCPv6 Advertisement to the Client to the DHCPv6 port (546).

The HTTP Boot Client will receive multiple advertisements from different sources of DHCPv6 Services, possibly from DHCPv6 Services which recognize the HTTP extensions or from Standard DHCPv6 Services. A DHCPv6 service recognizing HTTP extensions must respond with an Advertisement that has Option 16 (OPTION_VENDOR_CLASS) parameter set to “HTTPClient”, in response to the OPTION_VENDOR_CLASS requested in Option 6 in the DHCPv6 Solicit message.

Each message contains standard DHCP parameters: Identify Association (IA) option which conveys information including <IP address, lifetime, etc...>. Name server option conveys the DNS server address. The DHCP service or Proxy DHCP which recognizes the HTTPBoot extension will provide DHCPv6 Advertisement with HTTPClient extensions, including Boot URI and Optional Boot Parameters.

From the received DHCPOFFER(s), the client records the information as follows:
• Client IP address (and other parameters) provide through IA option
• Boot URI provided thru option 59
• Optional BootFile Parameter provided through option 60 (if no other parameter needed for this boot URI, this option can be eliminated)
• Optional Name-server information provided through option 23, if URI is displayed using domain-name.

24.7.5.1.3 Client multicast DHCPv6 Request to the selected DHCP Advertisement to confirm the IP address assigned by that server

This packet is the same with the DHCPv6 Solicit packet except for the message type is Request.

24.7.5.1.4 Server unicast the DHCPv6 Reply to acknowledge the Client IP address for the UEFI HTTP Client.

24.7.5.2 Message exchange between UEFI Client and DHCPv6 server not using DHCP Client Extensions

In a home environment, the Boot URI Information will not be provided by the DHCPv6 Offers, we need other channels to provide this information. Like what is described in Section 24.7.4.2, the implementation suggestion is provisioning this information by OEM or input by end user through Setup Options, henceforth, the UEFI Boot Client already know the Boot URI before contacting the DHCP server.

The message exchange between the EFI Client and DHCP server will be standard DHCP S.A.R.R. to obtain <IP address, Name-server>.
In the case of a home environment without a DHCPv6 server, the above message exchange is not needed, and the UEFI HTTP Boot Client will have the <IP address, Name-server> provisioned by OEM or input by the end user through Setup Options.

24.7.5.3 Message exchange between UEFI Client and DNS6 server
The DNS Query/Reply for domain name resolution is the same process as described in Section 24.7.4.3.

24.7.5.4 Message in HTTP Download
HTTP Download process is the same process as described in Section 24.7.4.4.

24.7.6 EFI HTTP Boot Callback Protocol
This section defines the EFI HTTP Boot Callback Protocol that is invoked when the HTTP Boot driver is about to transmit or has received a packet. The EFI HTTP Boot Callback Protocol must be installed on the same handle as the Load File Protocol for the HTTP Boot.

EFI_HTTP_BOOT_CALLBACK_PROTOCOL

Summary
Protocol that is invoked when the HTTP Boot driver is about to transmit or has received a packet.

GUID
#define EFI_HTTP_BOOT_CALLBACK_PROTOCOL_GUID
{0xba23b311, 0x343d, 0x11e6, {0x91, 0x85, 0x58, 0x20, 0xb1, 0xd6, 0x52, 0x99}}

Protocol Interface Structure
typedef struct _EFI_HTTP_BOOT_CALLBACK_PROTOCOL {
    EFI_HTTP_BOOT_CALLBACK       Callback;
} EFI_HTTP_BOOT_CALLBACK_PROTOCOL;

Parameters
Callback                Callback routine used by the HTTP Boot driver.

EFI_HTTP_BOOT_CALLBACK_PROTOCOL.Callback()

Summary
Callback function that is invoked when the HTTP Boot driver is about to transmit or has received a packet.

Prototype
typedef
EFI_STATUS
(EIFIAPI *EFI_HTTP_BOOT_CALLBACK) (
    IN EFI_HTTP_BOOT_CALLBACK_PROTOCOL*This,
IN EFI_HTTP_BOOT_CALLBACK_DATA_TYPE DataType,
IN BOOLEAN Received,
IN UINT32 DataLength,
IN VOID *Data OPTIONAL
);

Parameters
This Pointer to the EFI_HTTP_BOOT_CALLBACK_PROTOCOL instance.
DataType The event that occurs in the current state. Type EFI_HTTP_BOOT_CALLBACK_DATA_TYPE is defined below.
Received TRUE if the callback is being invoked due to a receive event. FALSE if the callback is being invoked due to a transmit event.
DataLength The length in bytes of the buffer pointed to by Data.
Data A pointer to the buffer of data, the data type is specified by DataType.

Related Definitions
//**********************************************
// EFI_HTTP_BOOT_CALLBACK_DATA_TYPE
//**********************************************
typedef enum {
    HttpBootDhcp4,
    HttpBootDhcp6,
    HttpBootHttpRequest,
    HttpBootHttpResponse,
    HttpBootHttpEntityBody,
    HttpBootTypeMax
} EFI_HTTP_BOOT_CALLBACK_DATA_TYPE;

HttpBootDhcp4 Data points to a DHCP4 packet which is about to transmit or has received.
HttpBootDhcp6 Data points to a DHCP6 packet which is about to be transmit or has received.
HttpBootHttpRequestData points to an EFI_HTTP_MESSAGE structure, which contains a HTTP request message to be transmitted.
HttpBootHttpResponseData points to an EFI_HTTP_MESSAGE structure, which contains a received HTTP response message.
HttpBootHttpEntityBody Part of the entity body has been received from the HTTP server. Data points to the buffer of the entity body data.

Description
This function is invoked when the HTTP Boot driver is about to transmit or has received packet. Parameters DataType and Received specify the type of event and the format of the buffer pointed to by Data. Due to the polling nature of UEFI device drivers, this callback function should not execute for more than 5 ms.
The returned status code determines the behavior of the HTTP Boot driver.

**Status Codes Returned**

<table>
<thead>
<tr>
<th>Status Code</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>EFI_SUCCESS</td>
<td>Tells the HTTP Boot driver to continue the HTTP Boot process.</td>
</tr>
<tr>
<td>EFI_ABORTED</td>
<td>Tells the HTTP Boot driver to abort the current HTTP Boot process.</td>
</tr>
</tbody>
</table>
25 - Network Protocols — Managed Network

25.1 EFI Managed Network Protocol

This chapter defines the EFI Managed Network Protocol. It is split into the following two main sections:

- Managed Network Service Binding Protocol (MNSBP)
- Managed Network Protocol (MNP)

The MNP provides raw (unformatted) asynchronous network packet I/O services. These services make it possible for multiple-event-driven drivers and applications to access and use the system network interfaces at the same time.

EFI_MANAGED_NETWORK_SERVICE_BINDING_PROTOCOL

Summary

The MNSBP is used to locate communication devices that are supported by an MNP driver and to create and destroy instances of the MNP child protocol driver that can use the underlying communications device.

The EFI Service Binding Protocol in Section 2.5.8 defines the generic Service Binding Protocol functions. This section discusses the details that are specific to the MNP.

GUID

```
#define EFI_MANAGED_NETWORK_SERVICE_BINDING_PROTOCOL_GUID \
{0xf36ff770,0xa7e1,0x42cf,\ 
 {0x9e,0xd2,0x56,0xf0,0xf2,0x71,0xf4,0x4c}}
```

Description

A network application (or driver) that requires shared network access can use one of the protocol handler services, such as `BS->LocateHandleBuffer()`, to search for devices that publish an MNSBP GUID. Each device with a published MNSBP GUID supports MNP and may be available for use.

After a successful call to the `EFI_MANAGED_NETWORK_SERVICE_BINDING_PROTOCOL.CreateChild()` function, the child MNP driver instance is in an unconfigured state; it is not ready to send and receive data packets.

Before a network application terminates execution, every successful call to the `EFI_MANAGED_NETWORK_SERVICE_BINDING_PROTOCOL.CreateChild()` function must be matched with a call to the `EFI_MANAGED_NETWORK_SERVICE_BINDING_PROTOCOL.DestroyChild()` function.

EFI_MANAGED_NETWORK_PROTOCOL

Summary

The MNP is used by network applications (and drivers) to perform raw (unformatted) asynchronous network packet I/O.
GUID

```c
#define EFI_MANAGED_NETWORK_PROTOCOL_GUID
    {0x7ab33a91, 0xace5, 0x4326,
     {0xb5, 0x72, 0xe7, 0xee, 0x33, 0xd3, 0x9f, 0x16}}
```

Protocol Interface Structure

```c
typedef struct _EFI_MANAGED_NETWORK_PROTOCOL {
    EFI_MANAGED_NETWORK_GET_MODE_DATA GetModeData;
    EFI_MANAGED_NETWORK_CONFIGURE Configure;
    EFI_MANAGED_NETWORK_MCAST_IP_TO_MAC McastIpToMac;
    EFI_MANAGED_NETWORK_GROUPS Groups;
    EFI_MANAGED_NETWORK_TRANSMIT Transmit;
    EFI_MANAGED_NETWORK_RECEIVE Receive;
    EFI_MANAGED_NETWORK_CANCEL Cancel;
    EFI_MANAGED_NETWORK_POLL Poll;
} EFI_MANAGED_NETWORK_PROTOCOL;
```

Parameters

- **GetModeData**: Returns the current MNP child driver operational parameters. May also support returning underlying Simple Network Protocol (SNP) driver mode data. See the `GetModeData()` function description.
- **Configure**: Sets and clears operational parameters for an MNP child driver. See the `Configure()` function description.
- **McastIpToMac**: Translates a software (IP) multicast address to a hardware (MAC) multicast address. This function may be unsupported in some MNP implementations. See the `McastIpToMac()` function description.
- **Groups**: Enables and disables receive filters for multicast addresses. This function may be unsupported in some MNP implementations. See the `Groups()` function description.
- **Transmit**: Places asynchronous outgoing data packets into the transmit queue. See the `Transmit()` function description.
- **Receive**: Places an asynchronous receiving request into the receiving queue. See the `Receive()` function description.
- **Cancel**: Aborts a pending transmit or receive request. See the `Cancel()` function description.
- **Poll**: Polls for incoming data packets and processes outgoing data packets. See the `Poll()` function description.

Description

The services that are provided by MNP child drivers make it possible for multiple drivers and applications to send and receive network traffic using the same network device.
Before any network traffic can be sent or received, the
`EFI_MANAGED_NETWORK_PROTOCOL.Configure()` function must initialize the operational
parameters for the MNP child driver instance. Once configured, data packets can be received and sent
using the following functions:

- `EFI_MANAGED_NETWORK_PROTOCOL.Transmit()`
- `EFI_MANAGED_NETWORK_PROTOCOL.Receive()`
- `EFI_MANAGED_NETWORK_PROTOCOL.Poll()`

**EFI_MANAGED_NETWORK_PROTOCOL.GetModeData()**

**Summary**
Returns the operational parameters for the current MNP child driver. May also support returning the
underlying SNP driver mode data.

**Prototype**

```c
typedef
EFI_STATUS
(EIFIAPI *EFI_MANAGED_NETWORK_GET_MODE_DATA) (  
    IN EFI_MANAGED_NETWORK_PROTOCOL *This,
    OUT EFI_MANAGED_NETWORK_CONFIG_DATA *MnpConfigData OPTIONAL,
    OUT EFI_SIMPLE_NETWORK_MODE *SnpModeData OPTIONAL
  )
;
```

**Parameters**

- **This**
  Pointer to the `EFI_MANAGED_NETWORK_PROTOCOL` instance.
- **MnpConfigData**
  Pointer to storage for MNP operational parameters. Type
  `EFI_MANAGED_NETWORK_CONFIG_DATA` is defined in “Related
  Definitions” below.
- **SnpModeData**
  Pointer to storage for SNP operational parameters. This
  feature may be unsupported. Type
  `EFI_SIMPLE_NETWORK_MODE` is defined in the
  `EFI_SIMPLE_NETWORK_PROTOCOL`.

**Description**
The `GetModeData()` function is used to read the current mode data (operational parameters) from the
MNP or the underlying SNP.
ReceivedQueueTimeoutValue
Timeout value for a UEFI one-shot timer event. A packet that has not been removed from the MNP receive queue by a call to `EFI_MANAGED_NETWORK_PROTOCOL.Poll()` will be dropped if its receive timeout expires. If this value is zero, then there is no receive queue timeout. If the receive queue fills up, then the device receive filters are disabled until there is room in the receive queue for more packets. The startup default value is 10,000,000 (10 seconds).

TransmitQueueTimeoutValue
Timeout value for a UEFI one-shot timer event. A packet that has not been removed from the MNP transmit queue by a call to `EFI_MANAGED_NETWORK_PROTOCOL.Poll()` will be dropped if its transmit timeout expires. If this value is zero, then there is no transmit queue timeout. If the transmit queue fills up, then the `EFI_MANAGED_NETWORK_PROTOCOL.Transmit()` function will return `EFI_NOT_READY` until there is room in the transmit queue for more packets. The startup default value is 10,000,000 (10 seconds).

ProtocolTypeFilter
Ethernet type II 16-bit protocol type in host byte order. Valid values are zero and 1,500 to 65,535. Set to zero to receive packets with any protocol type. The startup default value is zero.

EnableUnicastReceive
Set to `TRUE` to receive packets that are sent to the network device MAC address. The startup default value is `FALSE`. 
EnableMulticastReceive  
Set to TRUE to receive packets that are sent to any of the active multicast groups. The startup default value is FALSE.

EnableBroadcastReceive  
Set to TRUE to receive packets that are sent to the network device broadcast address. The startup default value is FALSE.

EnablePromiscuousReceive  
Set to TRUE to receive packets that are sent to any MAC address. Note that setting this field to TRUE may cause packet loss and degrade system performance on busy networks. The startup default value is FALSE.

FlushQueuesOnReset  
Set to TRUE to drop queued packets when the configuration is changed. The startup default value is FALSE.

EnableReceiveTimestamps  
Set to TRUE to timestamp all packets when they are received by the MNP. Note that timestamps may be unsupported in some MNP implementations. The startup default value is FALSE.

DisableBackgroundPolling  
Set to TRUE to disable background polling in this MNP instance. Note that background polling may not be supported in all MNP implementations. The startup default value is FALSE, unless background polling is not supported.

Status Codes Returned

<table>
<thead>
<tr>
<th>Status Code</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>EFI_SUCCESS</td>
<td>The operation completed successfully.</td>
</tr>
<tr>
<td>EFI_INVALID_PARAMETER</td>
<td>This is NULL.</td>
</tr>
<tr>
<td>EFI_UNSUPPORTED</td>
<td>The requested feature is unsupported in this MNP implementation.</td>
</tr>
<tr>
<td>EFI_NOT_STARTED</td>
<td>This MNP child driver instance has not been configured. The default values</td>
</tr>
<tr>
<td>Other</td>
<td>The mode data could not be read.</td>
</tr>
</tbody>
</table>

**EFI_MANAGED_NETWORK_PROTOCOL.Configure()**

**Summary**

Sets or clears the operational parameters for the MNP child driver.
Prototype

typedef

EFI_STATUS

(EIFIAPI *EFI_MANAGED_NETWORK_CONFIGURE) (

IN EFI_MANAGED_NETWORK_PROTOCOL *This,

IN EFI_MANAGED_NETWORK_CONFIG_DATA *MnpConfigData OPTIONAL

);

Parameters

This Pointer to the EFI_MANAGED_NETWORK_PROTOCOL instance.

MnpConfigData Pointer to configuration data that will be assigned to the MNP child driver instance. If NULL, the MNP child driver instance is reset to startup defaults and all pending transmit and receive requests are flushed. Type EFI_MANAGED_NETWORK_CONFIG_DATA is defined in EFI_MANAGED_NETWORK_PROTOCOL.GetModeData().

Description

The Configure() function is used to set, change, or reset the operational parameters for the MNP child driver instance. Until the operational parameters have been set, no network traffic can be sent or received by this MNP child driver instance. Once the operational parameters have been reset, no more traffic can be sent or received until the operational parameters have been set again.

Each MNP child driver instance can be started and stopped independently of each other by setting or resetting their receive filter settings with the Configure() function.

After any successful call to Configure(), the MNP child driver instance is started. The internal periodic timer (if supported) is enabled. Data can be transmitted and may be received if the receive filters have also been enabled.

Note: If multiple MNP child driver instances will receive the same packet because of overlapping receive filter settings, then the first MNP child driver instance will receive the original packet and additional instances will receive copies of the original packet.

Note: Warning: Receive filter settings that overlap will consume extra processor and/or DMA resources and degrade system and network performance.
### Status Codes Returned

<table>
<thead>
<tr>
<th>Code</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>EFI_SUCCESS</td>
<td>The operation completed successfully.</td>
</tr>
<tr>
<td>EFI_INVALID_PARAMETER</td>
<td>One or more of the following conditions is <strong>TRUE:</strong></td>
</tr>
<tr>
<td></td>
<td>- This is <strong>NULL</strong>.</td>
</tr>
<tr>
<td></td>
<td>- MnpConfigData.ProtocolTypeFilter is not valid.</td>
</tr>
<tr>
<td></td>
<td>The operational data for the MNP child driver instance is unchanged.</td>
</tr>
<tr>
<td>EFI_OUT_OF_RESOURCES</td>
<td>Required system resources (usually memory) could not be allocated.</td>
</tr>
<tr>
<td></td>
<td>The MNP child driver instance has been reset to startup defaults.</td>
</tr>
<tr>
<td>EFI_UNSUPPORTED</td>
<td>The requested feature is unsupported in this [MNP] implementation.</td>
</tr>
<tr>
<td></td>
<td>The operational data for the MNP child driver instance is unchanged.</td>
</tr>
<tr>
<td>EFI_DEVICE_ERROR</td>
<td>An unexpected network or system error occurred.</td>
</tr>
<tr>
<td></td>
<td>The MNP child driver instance has been reset to startup defaults.</td>
</tr>
<tr>
<td>Other</td>
<td>The MNP child driver instance has been reset to startup defaults.</td>
</tr>
</tbody>
</table>

### EFI_MANAGED_NETWORK_PROTOCOL.McastIpToMac()

#### Summary
Translates an IP multicast address to a hardware (MAC) multicast address. This function may be unsupported in some MNP implementations.

#### Prototype

```c
typedef EFI_STATUS (EFIAPI *EFI_MANAGED_NETWORK_MCAST_IP_TO_MAC) (  
    IN EFI_MANAGED_NETWORK_PROTOCOL *This,  
    IN BOOLEAN Ipv6Flag,  
    IN EFI_IP_ADDRESS *IpAddress,  
    OUT EFI_MAC_ADDRESS *MacAddress  
);
```

#### Parameters

- **This** Pointer to the **EFI_MANAGED_NETWORK_PROTOCOL** instance.
- **Ipv6Flag** Set to **TRUE** if **IpAddress** is an IPv6 multicast address. Set to **FALSE** if **IpAddress** is an IPv4 multicast address.
- **IpAddress** Pointer to the multicast IP address (in network byte order) to convert.
- **MacAddress** Pointer to the resulting multicast MAC address.

#### Description
The **McastIpToMac()** function translates an IP multicast address to a hardware (MAC) multicast address.

This function may be implemented by calling the underlying **EFI_SIMPLE_NETWORK.MCastIpToMac()** function, which may also be unsupported in some MNP implementations.
Status Codes Returned

<table>
<thead>
<tr>
<th>Code</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>EFI_SUCCESS</td>
<td>The operation completed successfully.</td>
</tr>
<tr>
<td>EFI_INVALID_PARAMETER</td>
<td>One of the following conditions is TRUE:</td>
</tr>
<tr>
<td></td>
<td>• This is NULL.</td>
</tr>
<tr>
<td></td>
<td>• IpAddress is NULL.</td>
</tr>
<tr>
<td></td>
<td>• *IpAddress is not a valid multicast IP address.</td>
</tr>
<tr>
<td></td>
<td>• MacAddress is NULL.</td>
</tr>
<tr>
<td>EFI_NOT_STARTED</td>
<td>This MNP child driver instance has not been configured.</td>
</tr>
<tr>
<td>EFI_UNSUPPORTED</td>
<td>The requested feature is unsupported in this MNP implementation.</td>
</tr>
<tr>
<td>EFI_DEVICE_ERROR</td>
<td>An unexpected network or system error occurred.</td>
</tr>
<tr>
<td>Other</td>
<td>The address could not be converted.</td>
</tr>
</tbody>
</table>

**EFI_MANAGED_NETWORK_PROTOCOL.Groups()**

**Summary**
Enables and disables receive filters for multicast address. This function may be unsupported in some MNP implementations.

**Prototype**

```c
typedef EFI_STATUS
(EFIAPI *EFI_MANAGED_NETWORK_GROUPS) (  
    IN EFI_MANAGED_NETWORK_PROTOCOL  *This,  
    IN BOOLEAN            JoinFlag,  
    IN EFI_MAC_ADDRESS    *MacAddress OPTIONAL  
);
```

**Parameters**

- **This**
  Pointer to the EFI_MANAGED_NETWORK_PROTOCOL instance.
- **JoinFlag**
  Set to TRUE to join this multicast group.
  Set to FALSE to leave this multicast group.
- **MacAddress**
  Pointer to the multicast MAC group (address) to join or leave.

**Description**
The Groups() function only adds and removes multicast MAC addresses from the filter list. The MNP driver does not transmit or process Internet Group Management Protocol (IGMP) packets.
If JoinFlag is FALSE and MacAddress is NULL, then all joined groups are left.
Status Codes Returned

<table>
<thead>
<tr>
<th>Code</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>EFI_SUCCESS</td>
<td>The requested operation completed successfully.</td>
</tr>
<tr>
<td>EFI_INVALID_PARAMETER</td>
<td>One or more of the following conditions is <strong>TRUE</strong>:</td>
</tr>
<tr>
<td></td>
<td>• <strong>This</strong> is <strong>NULL</strong>.</td>
</tr>
<tr>
<td></td>
<td>• <strong>JoinFlag</strong> is <strong>TRUE</strong> and <strong>MacAddress</strong> is <strong>NULL</strong>.</td>
</tr>
<tr>
<td></td>
<td>• <em>MacAddress</em> is not a valid multicast MAC address.</td>
</tr>
<tr>
<td></td>
<td>The MNP multicast group settings are unchanged.</td>
</tr>
<tr>
<td>EFI_NOT_STARTED</td>
<td>This MNP child driver instance has not been configured.</td>
</tr>
<tr>
<td>EFI_ALREADY_STARTED</td>
<td>The supplied multicast group is already joined.</td>
</tr>
<tr>
<td>EFI_NOT_FOUND</td>
<td>The supplied multicast group is not joined.</td>
</tr>
<tr>
<td>EFI_DEVICE_ERROR</td>
<td>An unexpected network or system error occurred.</td>
</tr>
<tr>
<td></td>
<td>The MNP child driver instance has been reset to startup defaults.</td>
</tr>
<tr>
<td>EFI_UNSUPPORTED</td>
<td>The requested feature is unsupported in this MNP implementation.</td>
</tr>
<tr>
<td>Other</td>
<td>The requested operation could not be completed.</td>
</tr>
<tr>
<td></td>
<td>The MNP multicast group settings are unchanged.</td>
</tr>
</tbody>
</table>

**EFI_MANAGED_NETWORK_PROTOCOL.Transmit()**

**Summary**
Places asynchronous outgoing data packets into the transmit queue.

**Prototype**

```c
typedef
EFI_STATUS
(EIFIAPI *EFI_MANAGED_NETWORK_TRANSMIT) (  
    IN EFI_MANAGED_NETWORK_PROTOCOL  *This,
    IN EFI_MANAGED_NETWORK_COMPLETION_TOKEN  *Token
);
```

**Parameters**

- **This**  
  Pointer to the **EFI_MANAGED_NETWORK_PROTOCOL** instance.

- **Token**  
  Pointer to a token associated with the transmit data descriptor. Type **EFI_MANAGED_NETWORK_COMPLETION_TOKEN** is defined in “Related Definitions” below.

**Description**

The **Transmit()** function places a completion token into the transmit packet queue. This function is always asynchronous.

The caller must fill in the **Token.Event** and **Token.TxData** fields in the completion token, and these fields cannot be **NULL**. When the transmit operation completes, the MNP updates the **Token.Status** field and the **Token.Event** is signaled.
**Note:** There may be a performance penalty if the packet needs to be defragmented before it can be transmitted by the network device. Systems in which performance is critical should review the requirements and features of the underlying communications device and drivers.

**Related Definitions**

```c
// *******************************************************
// EFI_MANAGED_NETWORK_COMPLETION_TOKEN
// *******************************************************
typedef struct {
    EFI_EVENT Event;
    EFI_STATUS Status;
    union {
        EFI_MANAGED_NETWORK_RECEIVE_DATA *RxData;
        EFI_MANAGED_NETWORK_TRANSMIT_DATA *TxData;
    }
    Packet;
} EFI_MANAGED_NETWORK_COMPLETION_TOKEN;
```

- **Event**: This *Event* will be signaled after the *Status* field is updated by the MNP. The type of *Event* must be *EVT_NOTIFY_SIGNAL*. The Task Priority Level (TPL) of *Event* must be lower than or equal to *TPL_CALLBACK*.

- **Status**: This field will be set to one of the following values:
  - *EFI_SUCCESS*: The receive or transmit completed successfully.
  - *EFI_ABORTED*: The receive or transmit was aborted.
  - *EFI_TIMEOUT*: The transmit timeout expired.
  - *EFI_DEVICE_ERROR*: There was an unexpected system or network error.
  - *EFI_NO_MEDIA*: There was a media error

- **RxData**: When this token is used for receiving, *RxData* is a pointer to the *EFI_MANAGED_NETWORK_RECEIVE_DATA*.

- **TxData**: When this token is used for transmitting, *TxData* is a pointer to the *EFI_MANAGED_NETWORK_TRANSMIT_DATA*.

The *EFI_MANAGED_NETWORK_COMPLETION_TOKEN* structure is used for both transmit and receive operations.

When it is used for transmitting, the *Event* and *TxData* fields must be filled in by the MNP client. After the transmit operation completes, the MNP updates the *Status* field and the *Event* is signaled.

When it is used for receiving, only the *Event* field must be filled in by the MNP client. After a packet is received, the MNP fills in the *RxData* and *Status* fields and the *Event* is signaled.
typedef struct {
  EFI_TIME   Timestamp;
  EFI_EVENT  RecycleEvent;
  UINT32     PacketLength;
  UINT32     HeaderLength;
  UINT32     AddressLength;
  UINT32     DataLength;
  BOOLEAN    BroadcastFlag;
  BOOLEAN    MulticastFlag;
  BOOLEAN    PromiscuousFlag;
  UINT16     ProtocolType;
  VOID       *DestinationAddress;
  VOID       *SourceAddress;
  VOID       *MediaHeader;
  VOID       *PacketData;
} EFI_MANAGED_NETWORK_RECEIVE_DATA;

Timestamp       System time when the MNP received the packet. Timestamp is zero filled if receive timestamps are disabled or unsupported.
RecycleEvent    MNP clients must signal this event after the received data has been processed so that the receive queue storage can be reclaimed. Once RecycleEvent is signaled, this structure and the received data that is pointed to by this structure must not be accessed by the client.
PacketLength    Length of the entire received packet (media header plus the data).
HeaderLength    Length of the media header in this packet.
AddressLength   Length of a MAC address in this packet.
DataLength      Length of the data in this packet.
BroadcastFlag   Set to TRUE if this packet was received through the broadcast filter. (The destination MAC address is the broadcast MAC address.)
MulticastFlag   Set to TRUE if this packet was received through the multicast filter. (The destination MAC address is in the multicast filter list.)
PromiscuousFlag Set to TRUE if this packet was received through the promiscuous filter. (The destination address does not match any of the other hardware or software filter lists.)
ProtocolType    16-bit protocol type in host byte order. Zero if there is no protocol type field in the packet header.
DestinationAddress Pointer to the destination address in the media header.
SourceAddress  Pointer to the source address in the media header.
MediaHeader    Pointer to the first byte of the media header.
PacketData     Pointer to the first byte of the packet data (immediately following media header).

An EFI_MANAGED_NETWORK_RECEIVE_DATA structure is filled in for each packet that is received by the MNP.

If multiple instances of this MNP driver can receive a packet, then the receive data structure and the received packet are duplicated for each instance of the MNP driver that can receive the packet.

```c
// EFI_MANAGED_NETWORK_TRANSMIT_DATA
typedef struct {
  EFI_MAC_ADDRESS *DestinationAddress OPTIONAL;
  EFI_MAC_ADDRESS *SourceAddress OPTIONAL;
  UINT16 ProtocolType OPTIONAL;
  UINT32 DataLength;
  UINT16 HeaderLength OPTIONAL;
  UINT16 FragmentCount;
  EFI_MANAGED_NETWORK_FRAGMENT_DATA FragmentTable[1];
} EFI_MANAGED_NETWORK_TRANSMIT_DATA;
```

DestinationAddress Pointer to the destination MAC address if the media header is not included in FragmentTable[]. If NULL, then the media header is already filled in FragmentTable[].
SourceAddress Pointer to the source MAC address if the media header is not included in FragmentTable[]. Ignored if DestinationAddress is NULL.
ProtocolType The protocol type of the media header in host byte order. Ignored if DestinationAddress is NULL.
DataLength Sum of all FragmentLength fields in FragmentTable[] minus the media header length.
HeaderLength Length of the media header if it is included in the FragmentTable. Must be zero if DestinationAddress is not NULL.
FragmentCount Number of data fragments in FragmentTable[]. This field cannot be zero.
FragmentTable Table of data fragments to be transmitted. The first byte of the first entry in FragmentTable[] is also the first byte of the media header or, if there is no media header, the first byte of payload. Type EFI_MANAGED_NETWORK_FRAGMENT_DATA is defined below.
The **EFI_MANAGED_NETWORK_TRANSMIT_DATA** structure describes a (possibly fragmented) packet to be transmitted.

The `DataLength` field plus the `HeaderLength` field must be equal to the sum of all of the `FragmentLength` fields in the `FragmentTable`.

If the media header is included in `FragmentTable[]`, then it cannot be split between fragments.

```c
typedef struct {
    UINT32    FragmentLength;
    VOID      *FragmentBuffer;
} EFI_MANAGED_NETWORK_FRAGMENT_DATA;
```

**FragmentLength**  Number of bytes in the `FragmentBuffer`. This field may not be set to zero.

**FragmentBuffer**  Pointer to the fragment data. This field may not be set to NULL.

The **EFI_MANAGED_NETWORK_FRAGMENT_DATA** structure describes the location and length of a packet fragment to be transmitted.

**Status Codes Returned**

<table>
<thead>
<tr>
<th>Code</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>EFI_SUCCESS</td>
<td>The transmit completion token was cached.</td>
</tr>
<tr>
<td>EFI_NOT_STARTED</td>
<td>This MNP child driver instance has not been configured.</td>
</tr>
<tr>
<td>EFI_INVALID_PARAMETER</td>
<td>One or more of the following conditions is <strong>TRUE</strong>:</td>
</tr>
<tr>
<td></td>
<td>• <strong>This</strong> is <strong>NULL</strong>.</td>
</tr>
<tr>
<td></td>
<td>• <strong>Token</strong> is <strong>NULL</strong>.</td>
</tr>
<tr>
<td></td>
<td>• <strong>Token.Event</strong> is <strong>NULL</strong>.</td>
</tr>
<tr>
<td></td>
<td>• <strong>Token.TxData</strong> is <strong>NULL</strong>.</td>
</tr>
<tr>
<td></td>
<td>• <strong>Token.TxData.DestinationAddress</strong> is not <strong>NULL</strong> and <strong>Token.TxData.HeaderLength</strong> is zero.</td>
</tr>
<tr>
<td></td>
<td>• <strong>Token.TxData.FragmentCount</strong> is zero.</td>
</tr>
<tr>
<td></td>
<td>• (**Token.TxData.HeaderLength + <strong>Token.TxData.DataLength</strong>) is not equal to the sum of the <strong>Token.TxData.FragmentTable[].FragmentLength</strong> fields.</td>
</tr>
<tr>
<td></td>
<td>• One or more of the <strong>Token.TxData.FragmentTable[].FragmentLength</strong> fields is zero.</td>
</tr>
<tr>
<td></td>
<td>• One or more of the <strong>Token.TxData.FragmentTable[].FragmentBufferData</strong> fields is <strong>NULL</strong>.</td>
</tr>
<tr>
<td></td>
<td>• <strong>Token.TxData.DataLength</strong> is greater than MTU.</td>
</tr>
</tbody>
</table>
EFI_MANAGED_NETWORK_PROTOCOL.Receive()

Summary
Places an asynchronous receiving request into the receiving queue.

Prototype
```c
typedef
EFI_STATUS
(EIFIAPI *EFI_MANAGED_NETWORK_RECEIVE) (  
  IN EFI_MANAGED_NETWORK_PROTOCOL *This,
  IN EFI_MANAGED_NETWORK_COMPLETION_TOKEN *Token
);
```

Parameters
- **This**: Pointer to the EFI_MANAGED_NETWORK_PROTOCOL instance.
- **Token**: Pointer to a token associated with the receive data descriptor. Type EFI_MANAGED_NETWORK_COMPLETION_TOKEN is defined in EFI_MANAGED_NETWORK_PROTOCOL.Transmit().

Description
The Receive() function places a completion token into the receive packet queue. This function is always asynchronous.

The caller must fill in the Token.Event field in the completion token, and this field cannot be NULL. When the receive operation completes, the MNP updates the Token.Status and Token.RxData fields and the Token.Event is signaled.
Status Codes Returned

<table>
<thead>
<tr>
<th>Code</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>EFI_SUCCESS</td>
<td>The receive completion token was cached.</td>
</tr>
<tr>
<td>EFI_NOT_STARTED</td>
<td>This MNP child driver instance has not been configured.</td>
</tr>
<tr>
<td>EFI_INVALID_PARAMETER</td>
<td>One or more of the following conditions is <strong>true</strong>:</td>
</tr>
<tr>
<td></td>
<td>• This is <strong>null</strong></td>
</tr>
<tr>
<td></td>
<td>• Token is <strong>null</strong></td>
</tr>
<tr>
<td></td>
<td>• Token.Event is <strong>null</strong></td>
</tr>
<tr>
<td>EFI_OUT_OF_RESOURCES</td>
<td>The transmit data could not be queued due to a lack of system resources (usually memory).</td>
</tr>
<tr>
<td>EFI_DEVICE_ERROR</td>
<td>An unexpected system or network error occurred. The MNP child driver instance has been reset to startup defaults.</td>
</tr>
<tr>
<td>EFI_ACCESS_DENIED</td>
<td>The receive completion token was already in the receive queue.</td>
</tr>
<tr>
<td>EFI_NOT_READY</td>
<td>The receive request could not be queued because the receive queue is full.</td>
</tr>
<tr>
<td>EFI_NO_MEDIA</td>
<td>There was a media error.</td>
</tr>
</tbody>
</table>

**EFI_MANAGED_NETWORK_PROTOCOL.Cancel()**

**Summary**

Aborts an asynchronous transmit or receive request.

**Prototype**

```c
typedef
  EFI_STATUS
  (EFIAPI *EFI_MANAGED_NETWORK_CANCEL)(
    IN EFI_MANAGED_NETWORK_PROTOCOL *This,
    IN EFI_MANAGED_NETWORK_COMPLETION_TOKEN *Token OPTIONAL
  );
```

**Parameters**

- **This**
  Pointer to the **EFI_MANAGED_NETWORK_PROTOCOL** instance.
- **Token**
  Pointer to a token that has been issued by
  **EFI_MANAGED_NETWORK_PROTOCOL.Transmit()** or
  **EFI_MANAGED_NETWORK_PROTOCOL.Receive()**. If **NULL**, all pending tokens are aborted. Type
  **EFI_MANAGED_NETWORK_COMPLETION_TOKEN** is defined in
  **EFI_MANAGED_NETWORK_PROTOCOL.Transmit()**.

**Description**

The **Cancel()** function is used to abort a pending transmit or receive request. If the token is in the transmit or receive request queues, after calling this function, **Token.Status** will be set to **EFI_ABORTED** and then **Token.Event** will be signaled. If the token is not in one of the queues, which usually means that the asynchronous operation has completed, this function will not signal the token and **EFI_NOT_FOUND** is returned.
Status Codes Returned

<table>
<thead>
<tr>
<th>Code</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>EFI_SUCCESS</td>
<td>The asynchronous I/O request was aborted and <code>Token.Event</code> was signaled. When <code>Token</code> is NULL, all pending requests were aborted and their events were signaled.</td>
</tr>
<tr>
<td>EFI_NOT_STARTED</td>
<td>This MNP child driver instance has not been configured.</td>
</tr>
<tr>
<td>EFI_INVALID_PARAMETER</td>
<td>This is NULL.</td>
</tr>
<tr>
<td>EFI_NOT_FOUND</td>
<td>When <code>Token</code> is not NULL, the asynchronous I/O request was not found in the transmit or receive queue. It has either completed or was not issued by <code>Transmit()</code> and <code>Receive()</code>.</td>
</tr>
</tbody>
</table>

EFI_MANAGED_NETWORK_PROTOCOL.Poll()

Summary
Polls for incoming data packets and processes outgoing data packets.

Prototype
```c
typedef EFI_STATUS (EFIAPI *EFI_MANAGED_NETWORK_POLL) (
    IN EFI_MANAGED_NETWORK_PROTOCOL  *This
);
```

Parameters
This Pointer to the EFI_MANAGED_NETWORK_PROTOCOL instance.

Description
The Poll() function can be used by network drivers and applications to increase the rate that data packets are moved between the communications device and the transmit and receive queues.

Normally, a periodic timer event internally calls the Poll() function. But, in some systems, the periodic timer event may not call Poll() fast enough to transmit and/or receive all data packets without missing packets. Drivers and applications that are experiencing packet loss should try calling the Poll() function more often.

Status Codes Returned

<table>
<thead>
<tr>
<th>Code</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>EFI_SUCCESS</td>
<td>Incoming or outgoing data was processed.</td>
</tr>
<tr>
<td>EFI_NOT_STARTED</td>
<td>This MNP child driver instance has not been configured.</td>
</tr>
<tr>
<td>EFI_DEVICE_ERROR</td>
<td>An unexpected system or network error occurred. The MNP child driver instance has been reset to startup defaults.</td>
</tr>
<tr>
<td>EFI_NOT_READY</td>
<td>No incoming or outgoing data was processed. Consider increasing the polling rate.</td>
</tr>
<tr>
<td>EFI_TIMEOUT</td>
<td>Data was dropped out of the transmit and/or receive queue. Consider increasing the polling rate.</td>
</tr>
</tbody>
</table>
26 - Network Protocols — Bluetooth

26.1 EFI Bluetooth Host Controller Protocol

**EFI_BLUETOOTH_HC_PROTOCOL**

**Summary**

This protocol abstracts the Bluetooth host controller layer message transmit and receive.

**GUID**

```c
#define EFI_BLUETOOTH_HC_PROTOCOL_GUID \
{ 0xb3930571, 0xbeba, 0x4fc5, \
{ 0x92, 0x3, 0x94, 0x27, 0x24, 0x2e, 0x6a, 0x43 }}
```

**Protocol Interface Structure**

```c
typedef struct _EFI_BLUETOOTH_HC_PROTOCOL {
  EFI_BLUETOOTH_HC_SEND_COMMAND    SendCommand;
  EFI_BLUETOOTH_HC_RECEIVE_EVENT   ReceiveEvent;
  EFI_BLUETOOTH_HC_ASYNC_RECEIVE_EVENT  AsyncReceiveEvent;
  EFI_BLUETOOTH_HC_SEND_ACL_DATA   SendACLData;
  EFI_BLUETOOTH_HC_RECEIVE_ACL_DATA ReceiveACLData;
  EFI_BLUETOOTH_HC_ASYNC_RECEIVE_ACL_DATA AsyncReceiveACLData;
  EFI_BLUETOOTH_HC_SEND_SCO_DATA   SendSCOData;
  EFI_BLUETOOTH_HC_RECEIVE_SCO_DATA ReceiveSCOData;
  EFI_BLUETOOTH_HC_ASYNC_RECEIVE_SCO_DATA AsyncReceiveSCOData;
} EFI_BLUETOOTH_HC_PROTOCOL;
```

**Parameters**

- **SendCommand** Send HCI command packet. See the `SendCommand()` function description.
- **ReceiveEvent** Receive HCI event packets. See the `ReceiveEvent()` function description.
- **AsyncReceiveEvent** Non-blocking receive HCI event packets. See the `AsyncReceiveEvent()` function description.
- **SendACLData** Send HCI ACL (asynchronous connection-oriented) data packets. See the `SendACLData()` function description.
- **ReceiveACLData** Receive HCI ACL data packets. See the `ReceiveACLData()` function description.
- **AsyncReceiveACLData** Non-blocking receive HCI ACL data packets. See the `AsyncReceiveACLData()` function description.
- **SendSCOData** Send HCI synchronous (SCO and eSCO) data packets. See the `SendSCOData()` function description.
- **ReceiveSCOData** Receive HCI synchronous data packets. See the `ReceiveSCOData()` function description.
AsyncReceiveSCOData: Non-blocking receive HCI synchronous data packets. See the AsyncReceiveSCOData() function description.

Description
The EFI_BLUETOOTH_HC_PROTOCOL is used to transmit or receive HCI layer data packets. For detail of different HCI packet (command, event, ACL, SCO), please refer to Bluetooth specification.

BLUETOOTH_HC_PROTOCOL.SendCommand()

Summary
Send HCI command packet.

Prototype
typedef EFI_STATUS (EFIAPI *EFI_BLUETOOTH_HC_SEND_COMMAND)(
    IN EFI_BLUETOOTH_HC_PROTOCOL   *This,
    IN OUT UINTN           *BufferSize,
    IN VOID              *Buffer,
    IN UINTN            Timeout
);

Parameters
This Pointer to the EFI_BLUETOOTH_HC_PROTOCOL instance.
BufferSize On input, indicates the size, in bytes, of the data buffer specified by Buffer. On output, indicates the amount of data actually transferred.
Buffer A pointer to the buffer of data that will be transmitted to Bluetooth host controller.
Timeout Indicating the transfer should be completed within this time frame. The units are in milliseconds. If Timeout is 0, then the caller must wait for the function to be completed until EFI_SUCCESS or EFI_DEVICE_ERROR is returned.

Description
The SendCommand() function sends HCI command packet. Buffer holds the whole HCI command packet, including OpCode, OCF, OGF, parameter length, and parameters. When this function is returned, it just means the HCI command packet is sent, it does not mean the command is success or complete. Caller might need to wait a command status event to know the command status, or wait a command complete event to know if the command is completed. (see in Bluetooth specification, HCI Command Packet for more detail)
Status Codes Returned

<table>
<thead>
<tr>
<th>Code</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>EFI_SUCCESS</td>
<td>The HCI command packet is sent successfully.</td>
</tr>
<tr>
<td>EFI_INVALID_PARAMETER</td>
<td>One or more of the following conditions is TRUE:</td>
</tr>
<tr>
<td></td>
<td>BufferSize is NULL.</td>
</tr>
<tr>
<td></td>
<td>*BufferSize is 0.</td>
</tr>
<tr>
<td></td>
<td>Buffer is NULL.</td>
</tr>
<tr>
<td>EFI_TIMEOUT</td>
<td>Sending HCI command packet fail due to timeout.</td>
</tr>
<tr>
<td>EFI_DEVICE_ERROR</td>
<td>Sending HCI command packet fail due to host controller or device error.</td>
</tr>
</tbody>
</table>

**BLUETOOTH_HC_PROTOCOL.ReceiveEvent()**

**Summary**

Receive HCI event packet.

**Prototype**

```c
typedef EFI_STATUS
  (EFIAPI *EFI_BLUETOOTH_HC_RECEIVE_EVENT)(
   IN EFI_BLUETOOTH_HC_PROTOCOL *This,
   IN OUT UINTN *BufferSize,
   OUT VOID *Buffer,
   IN UINTN Timeout)
);
```

**Parameters**

- **This**
  - Pointer to the `EFI_BLUETOOTH_HC_PROTOCOL` instance.
- **BufferSize**
  - On input, indicates the size, in bytes, of the data buffer specified by `Buffer`. On output, indicates the amount of data actually transferred.
- **Buffer**
  - A pointer to the buffer of data that will be received from Bluetooth host controller.
- **Timeout**
  - Indicating the transfer should be completed within this time frame. The units are in milliseconds. If `Timeout` is 0, then the caller must wait for the function to be completed until `EFI_SUCCESS` or `EFI_DEVICE_ERROR` is returned.

**Description**

The `ReceiveEvent()` function receives HCI event packet. `Buffer` holds the whole HCI event packet, including `EventCode`, parameter length, and parameters. (See in Bluetooth specification, HCI Event Packet for more detail.)
Status Codes Returned

<table>
<thead>
<tr>
<th>Code</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>EFI_SUCCESS</td>
<td>The HCI event packet is received successfully.</td>
</tr>
<tr>
<td>EFI_INVALID_PARAMETER</td>
<td>One or more of the following conditions is TRUE:</td>
</tr>
<tr>
<td></td>
<td>• BufferSize is NULL.</td>
</tr>
<tr>
<td></td>
<td>• *BufferSize is 0.</td>
</tr>
<tr>
<td></td>
<td>• Buffer is NULL.</td>
</tr>
<tr>
<td>EFI_TIMEOUT</td>
<td>Receiving HCI event packet fail due to timeout.</td>
</tr>
<tr>
<td>EFI_DEVICE_ERROR</td>
<td>Receiving HCI event packet fail due to host controller or device error.</td>
</tr>
</tbody>
</table>

BLUETOOTH_HC_PROTOCOL.AsyncReceiveEvent()

Summary
Receive HCI event packet in non-blocking way.

Prototype
```c
typedef EFI_STATUS (EFIAPI *EFI_BLUETOOTH_HC_ASYNC_RECEIVE_EVENT) (  
  IN EFI_BLUETOOTH_HC_PROTOCOL       *This,  
  IN BOOLEAN                        IsNewTransfer,  
  IN UINTN                          PollingInterval,  
  IN UINTN                          DataLength,  
  IN EFI_BLUETOOTH_HC_ASYNC_FUNC_CALLBACK  Callback,  
  IN VOID                          *Context
);
```

Parameters
- **This**: Pointer to the `EFI_BLUETOOTH_HC_PROTOCOL` instance.
- **IsNewTransfer**: If `TRUE`, a new transfer will be submitted. If `FALSE`, the request is deleted.
- **PollingInterval**: Indicates the periodic rate, in milliseconds, that the transfer is to be executed.
- **DataLength**: Specifies the length, in bytes, of the data to be received.
- **Callback**: The callback function. This function is called if the asynchronous transfer is completed.
- **Context**: Data passed into `Callback` function. This is optional parameter and may be `NULL`.

Description
The `AsyncReceiveEvent()` function receives HCI event packet in non-blocking way. `Data` in `Callback` function holds the whole HCI event packet, including `EventCode`, parameter length, and parameters. (See in Bluetooth specification, HCI Event Packet for more detail.)
Related Definitions

typedef EFI_STATUS (EFIAPI *EFI_BLUETOOTH_HC_ASYNC_FUNC_CALLBACK) (
    IN VOID              *Data,
    IN UINTN             DataLength,
    IN VOID              *Context
);

Data    Data received via asynchronous transfer.
DataLength The length of Data in bytes, received via asynchronous transfer.
Context  Context passed from asynchronous transfer request.

Status Codes Returned

<table>
<thead>
<tr>
<th>Code</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>EFI_SUCCESS</td>
<td>The HCI asynchronous receive request is submitted successfully.</td>
</tr>
<tr>
<td>EFI_INVALID_PARAMETER</td>
<td>One or more of the following conditions is TRUE:</td>
</tr>
<tr>
<td></td>
<td>• DataLength is 0.</td>
</tr>
<tr>
<td></td>
<td>• If IsNewTransfer is TRUE, and an asynchronous receive request already exists.</td>
</tr>
</tbody>
</table>

BLUETOOTH_HC_PROTOCOL.SendACLData()

Summary
Send HCI ACL data packet.

Prototype

typedef EFI_STATUS (EFIAPI *EFI_BLUETOOTH_HC_SEND_ACL_DATA)(
    IN EFI_BLUETOOTH_HC_PROTOCOL   *This,
    IN OUT UINTN           *BufferSize,
    IN VOID              *Buffer,
    IN UINTN             Timeout
);

Parameters

This Pointer to the EFI_BLUETOOTH_HC_PROTOCOL instance.
BufferSize On input, indicates the size, in bytes, of the data buffer specified by Buffer. On output, indicates the amount of data actually transferred.
Buffer A pointer to the buffer of data that will be transmitted to Bluetooth host controller.
Timeout Indicating the transfer should be completed within this time frame. The units are in milliseconds. If Timeout is 0, then the caller must
wait for the function to be completed until \texttt{EFI\_SUCCESS} or \texttt{EFI\_DEVICE\_ERROR} is returned.

**Description**

The \texttt{SendACLData()} function sends HCI ACL data packet. \texttt{Buffer} holds the whole HCI ACL data packet, including Handle, PB flag, BC flag, data length, and data. (see in Bluetooth specification, HCI ACL Data Packet for more detail)

The \texttt{SendACLData()} function and \texttt{ReceiveACLData()} function just send and receive data payload from application layer. In order to protect the payload data, the Bluetooth bus is required to call \texttt{HCI\_Set\_Connection\_Encryption} command to enable hardware based encryption after authentication completed, according to pairing mode and host capability.

**Status Codes Returned**

<table>
<thead>
<tr>
<th>Status Code</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>\texttt{EFI_SUCCESS}</td>
<td>The HCI ACL data packet is sent successfully.</td>
</tr>
<tr>
<td>\texttt{EFI_INVALID_PARAMETER}</td>
<td>One or more of the following conditions is TRUE:</td>
</tr>
<tr>
<td></td>
<td>• \texttt{BufferSize} is NULL.</td>
</tr>
<tr>
<td></td>
<td>• *\texttt{BufferSize} is 0.</td>
</tr>
<tr>
<td></td>
<td>• \texttt{Buffer} is NULL.</td>
</tr>
<tr>
<td>\texttt{EFI_TIMEOUT}</td>
<td>Sending HCI ACL data packet fail due to timeout.</td>
</tr>
<tr>
<td>\texttt{EFI_DEVICE_ERROR}</td>
<td>Sending HCI ACL data packet fail due to host controller or device error.</td>
</tr>
</tbody>
</table>

**BLUETOOTH\_HC\_PROTOCOL.ReceiveACLData()**

**Summary**

Receive HCI ACL data packet.

**Prototype**

```c
typedef
EFI\_STATUS
(\texttt{EFIAPI *EFI\_BLUETOOTH\_HC\_RECEIVE\_ACL\_DATA)}(
  \texttt{IN EFI\_BLUETOOTH\_HC\_PROTOCOL} *This,
  \texttt{IN OUT UINTN} *BufferSize,
  \texttt{OUT VOID} *Buffer,
  \texttt{IN UINTN} Timeout
);
```

**Parameters**

- **This**
  - Pointer to the \texttt{EFI\_BLUETOOTH\_HC\_PROTOCOL} instance.

- **BufferSize**
  - On input, indicates the size, in bytes, of the data buffer specified by \texttt{Buffer}. On output, indicates the amount of data actually transferred.

- **Buffer**
  - A pointer to the buffer of data that will be received from Bluetooth host controller.
Timeout

Indicating the transfer should be completed within this time frame. The units are in milliseconds. If Timeout is 0, then the caller must wait for the function to be completed until EFI_SUCCESS or EFI_DEVICE_ERROR is returned.

Description

The ReceiveACLData() function receives HCI ACL data packet. Buffer holds the whole HCI ACL data packet, including Handle, PB flag, BC flag, data length, and data. (See in Bluetooth specification, HCI ACL Data Packet for more detail.)

Status Codes Returned

<table>
<thead>
<tr>
<th>Status Code</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>EFI_SUCCESS</td>
<td>The HCI ACL data packet is received successfully.</td>
</tr>
<tr>
<td>EFI_INVALID_PARAMETER</td>
<td>One or more of the following conditions is TRUE:</td>
</tr>
<tr>
<td></td>
<td>• BufferSize is NULL.</td>
</tr>
<tr>
<td></td>
<td>• *BufferSize is 0.</td>
</tr>
<tr>
<td></td>
<td>• Buffer is NULL.</td>
</tr>
<tr>
<td>EFI_TIMEOUT</td>
<td>Receiving HCI ACL data packet fail due to timeout.</td>
</tr>
<tr>
<td>EFI_DEVICE_ERROR</td>
<td>Receiving HCI ACL data packet fail due to host controller or device error.</td>
</tr>
</tbody>
</table>

BLUETOOTH_HC_PROTOCOL.AsyncReceiveACLData()

Summary

Receive HCI ACL data packet in non-blocking way.

Prototype

typedef
EFI_STATUS
(EFIAPI *EFI_BLUETOOTH_HC_ASYNC_RECEIVE_ACL_DATA) (
   IN EFI_BLUETOOTH_HC_PROTOCOL       *This,
   IN BOOLEAN                     IsNewTransfer,
   IN UINTN                       PollingInterval,
   IN UINTN                       DataLength,
   IN EFI_BLUETOOTH_HC_ASYNC_FUNC_CALLBACK  Callback,
   IN VOID                        *Context
);

Parameters

This Pointer to the EFI_BLUETOOTH_HC_PROTOCOL instance.
IsNewTransfer If TRUE, a new transfer will be submitted. If FALSE, the request is deleted.
PollingInterval Indicates the periodic rate, in milliseconds, that the transfer is to be executed.
DataLength Specifies the length, in bytes, of the data to be received.
Callback
The callback function. This function is called if the asynchronous transfer is completed.

Context
Data passed into Callback function. This is optional parameter and may be NULL.

Description
The AsyncReceiveACLData() function receives HCI ACL data packet in non-blocking way. Data in Callback holds the whole HCI ACL data packet, including Handle, PB flag, BC flag, data length, and data. (See in Bluetooth specification, HCI ACL Data Packet for more detail.)

Status Codes Returned

<table>
<thead>
<tr>
<th>Status Code</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>EFI_SUCCESS</td>
<td>The HCI asynchronous receive request is submitted successfully.</td>
</tr>
<tr>
<td>EFI_INVALID_PARAMETER</td>
<td>One or more of the following conditions is TRUE:</td>
</tr>
<tr>
<td>• DataLength is 0.</td>
<td>• If IsNewTransfer is TRUE, and an asynchronous receive request already exists.</td>
</tr>
</tbody>
</table>

BLUETOOTH_HC_PROTOCOL.SendSCOData()

Summary
Send HCI SCO data packet.

Prototype

typedef
EFI_STATUS
(EIFIAPI *EFI_BLUETOOTH_HC_SEND_SCO_DATA)(
    IN EFI_BLUETOOTH_HC_PROTOCOL   *This,
    IN OUT UINTN           *BufferSize,
    IN VOID              *Buffer,
    IN UINTN            Timeout
);

Parameters

This
Pointer to the EFI_BLUETOOTH_HC_PROTOCOL instance.

BufferSize
On input, indicates the size, in bytes, of the data buffer specified by Buffer. On output, indicates the amount of data actually transferred.

Buffer
A pointer to the buffer of data that will be transmitted to Bluetooth host controller.

Timeout
Indicating the transfer should be completed within this time frame. The units are in milliseconds. If Timeout is 0, then the caller must wait for the function to be completed until EFI_SUCCESS or EFI_DEVICE_ERROR is returned.
Description
The `SendSCOData()` function sends HCI SCO data packet. `Buffer` holds the whole HCI SCO data packet, including `ConnectionHandle`, `PacketStatus` flag, data length, and data. (See in Bluetooth specification, HCI Synchronous Data Packet for more detail.)

Status Codes Returned

<table>
<thead>
<tr>
<th>Status Code</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>EFI_SUCCESS</td>
<td>The HCI SCO data packet is sent successfully.</td>
</tr>
<tr>
<td>EFI_UNSUPPORTED</td>
<td>The implementation does not support HCI SCO transfer.</td>
</tr>
<tr>
<td>EFI_INVALID_PARAMETER</td>
<td>One or more of the following conditions is TRUE:</td>
</tr>
<tr>
<td></td>
<td>• <code>BufferSize</code> is NULL.</td>
</tr>
<tr>
<td></td>
<td>• <code>*BufferSize</code> is 0.</td>
</tr>
<tr>
<td></td>
<td>• <code>Buffer</code> is NULL.</td>
</tr>
<tr>
<td>EFI_TIMEOUT</td>
<td>Sending HCI SCO data packet due to timeout.</td>
</tr>
<tr>
<td>EFI_DEVICE_ERROR</td>
<td>Sending HCI SCO data packet fail due to host controller or device error.</td>
</tr>
</tbody>
</table>

**BLUETOOTH_HC_PROTOCOL.ReceiveSCOData()**

Summary
Receive HCI SCO data packet.

Prototype
```c
typedef EFI_STATUS (EFIAPI *EFI_BLUETOOTH_HC_RECEIVE_SCO_DATA)(
    IN EFI_BLUETOOTH_HC_PROTOCOL   *This,
    IN OUT UINTN           *BufferSize,
    OUT VOID             *Buffer,
    IN UINTN                     Timeout
);
```

Parameters
- **This**
  Pointer to the `EFI_BLUETOOTH_HC_PROTOCOL` instance.
- **BufferSize**
  On input, indicates the size, in bytes, of the data buffer specified by `Buffer`. On output, indicates the amount of data actually transferred.
- **Buffer**
  A pointer to the buffer of data that will be received from Bluetooth host controller.
- **Timeout**
  Indicating the transfer should be completed within this time frame. The units are in milliseconds. If `Timeout` is 0, then the caller must wait for the function to be completed until `EFI_SUCCESS` or `EFI_DEVICE_ERROR` is returned.
Description

The `ReceiveSCOData()` function receives HCI SCO data packet. `Buffer` holds the whole HCI SCO data packet, including ConnectionHandle, PacketStatus flag, data length, and data. (see in Bluetooth specification, HCI Synchronous Data Packet for more detail)

Status Codes Returned

<table>
<thead>
<tr>
<th>Code</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>EFI_SUCCESS</td>
<td>The HCI SCO data packet is received successfully.</td>
</tr>
<tr>
<td>EFI_INVALID_PARAMETER</td>
<td>One or more of the following conditions is TRUE:</td>
</tr>
<tr>
<td></td>
<td>• BufferSize is NULL.</td>
</tr>
<tr>
<td></td>
<td>• *BufferSize is 0.</td>
</tr>
<tr>
<td></td>
<td>• Buffer is NULL.</td>
</tr>
<tr>
<td>EFI_TIMEOUT</td>
<td>Receiving HCI SCO data packet fail due to timeout.</td>
</tr>
<tr>
<td>EFI_DEVICE_ERROR</td>
<td>Receiving HCI SCO data packet fail due to host controller or device error.</td>
</tr>
</tbody>
</table>

**BLUETOOTH_HC_PROTOCOL.AsyncReceiveSCOData()**

Summary

Receive HCI SCO data packet in non-blocking way.

Prototype

```c
typedef EFI_STATUS
  (EFIAPI *EFI_BLUETOOTH_HC_ASYNC_RECEIVE_SCO_DATA) (
  IN EFI_BLUETOOTH_HC_PROTOCOL       *This,
  IN BOOLEAN                        IsNewTransfer,
  IN UINTN                          PollingInterval,
  IN UINTN                          DataLength,
  IN EFI_BLUETOOTH_HC_ASYNC_FUNC_CALLBACK Callback,
  IN VOID                           *Context
);
```

Parameters

- **This**
  Pointer to the `EFI_BLUETOOTH_HC_PROTOCOL` instance.
- **IsNewTransfer**
  If `TRUE`, a new transfer will be submitted. If `FALSE`, the request is deleted.
- **PollingInterval**
  Indicates the periodic rate, in milliseconds, that the transfer is to be executed.
- **DataLength**
  Specifies the length, in bytes, of the data to be received.
- **Callback**
  The callback function. This function is called if the asynchronous transfer is completed.
- **Context**
  Data passed into `Callback` function. This is optional parameter and may be `NULL`. 
Description

The `AsyncReceiveSCOData()` function receives HCI SCO data packet in non-blocking way. Data in `Callback` holds the whole HCI SCO data packet, including `ConnectionHandle`, `PacketStatus` flag, data length, and data. (See in Bluetooth specification, HCI SCO Data Packet for more detail.)

<table>
<thead>
<tr>
<th>Status Code</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>EFI_SUCCESS</td>
<td>The HCI asynchronous receive request is submitted successfully.</td>
</tr>
<tr>
<td>EFI_INVALID_PARAMETER</td>
<td>One or more of the following conditions is TRUE:</td>
</tr>
<tr>
<td></td>
<td>• <code>DataLength</code> is 0.</td>
</tr>
<tr>
<td></td>
<td>• If <code>IsNewTransfer</code> is TRUE, and an asynchronous receive request already exists.</td>
</tr>
</tbody>
</table>

### 26.2 EFI Bluetooth Bus Protocol

**EFI_BLUETOOTH_IO_SERVICE_BINDING_PROTOCOL**

**Summary**

The EFI Bluetooth IO Service Binding Protocol is used to locate EFI Bluetooth IO Protocol drivers to create and destroy child of the driver to communicate with other Bluetooth device by using Bluetooth IO protocol.

**GUID**

```c
#define EFI_BLUETOOTH_IO_SERVICE_BINDING_PROTOCOL_GUID \\  { 0x388278d3, 0x7b85, 0x42f0, \\  { 0xab, 0xa9, 0xfb, 0x4b, 0xfd, 0x69, 0xf5, 0xab }
```

**Description**

The Bluetooth IO consumer need locate `EFI_BLUETOOTH_IO_SERVICE_BINDING_PROTOCOL` and call `CreateChild()` to create a new child of `EFI_BLUETOOTH_IO_PROTOCOL` instance. Then use `EFI_BLUETOOTH_IO_PROTOCOL` for Bluetooth communication. After use, the Bluetooth IO consumer need call `DestroyChild()` to destroy it.

**EFI_BLUETOOTH_IO_PROTOCOL**

**Summary**

This protocol provides service for Bluetooth L2CAP (Logical Link Control and Adaptation Protocol) and SDP (Service Discovery Protocol).

**GUID**

```c
#define EFI_BLUETOOTH_IO_PROTOCOL_GUID \\  { 0x467313de, 0x4e30, 0x43f1, \\  { 0x94, 0x3e, 0x32, 0x3f, 0x89, 0x84, 0x5d, 0xb5 }}
```
Protocol Interface Structure

typedef struct _EFI_BLUETOOTH_IO_PROTOCOL {
    EFI_BLUETOOTH_IO_GET_DEVICE_INFO    GetDeviceInfo;
    EFI_BLUETOOTH_IO_GET_SDP_INFO      GetSdpInfo;
    EFI_BLUETOOTH_IO_L2CAP_RAW_SEND    L2CapRawSend;
    EFI_BLUETOOTH_IO_L2CAP_RAW_RECEIVE L2CapRawReceive;
    EFI_BLUETOOTH_IO_L2CAP_RAW_ASYNC_RECEIVE
        L2CapRawAsyncReceive;
    EFI_BLUETOOTH_IO_L2CAP_SEND        L2CapSend;
    EFI_BLUETOOTH_IO_L2CAP_RECEIVE     L2CapReceive;
    EFI_BLUETOOTH_IO_L2CAP_ASYNC_RECEIVE L2CapAsyncReceive;
    EFI_BLUETOOTH_IO_L2CAP_CONNECT     L2CapConnect;
    EFI_BLUETOOTH_IO_L2CAP_DISCONNECT  L2CapDisconnect;
    EFI_BLUETOOTH_IO_L2CAP_REGISTER_SERVICE
        L2CapRegisterService;
} EFI_BLUETOOTH_IO_PROTOCOL;

Parameters

GetDeviceInfo Get Bluetooth device Information. See the GetDeviceInfo() function description.

GetSdpInfo Get Bluetooth device SDP information. See the GetSdpInfo() function description.

L2CapRawSend Send L2CAP message (including L2CAP header). See the L2CapRawSend() function description.


L2CapSend Send L2CAP message (excluding L2CAP header) to a specific channel. See the L2CapSend() function description.

L2CapReceive Receive L2CAP message (excluding L2CAP header) from a specific channel. See the L2CapReceive() function description.

L2CapAsyncReceive Non-blocking receive L2CAP message (excluding L2CAP header) from a specific channel. See the L2CapAsyncReceive() function description.

L2CapConnect Do L2CAP connection. See the L2CapConnect() function description.

L2CapDisconnect Do L2CAP disconnection. See the L2CapDisconnect() function description.

L2CapRegisterService Register L2CAP callback function for special channel. See the L2CapRegisterService() function description.
Description
The **EFI_BLUETOOTH_IO_PROTOCOL** provides services in L2CAP protocol and SDP protocol. For detail of L2CAP packet format, and SDP service, please refer to Bluetooth specification.

**BLUETOOTH_IO_PROTOCOL.GetDeviceInfo**

**Summary**
Get Bluetooth device information.

**Prototype**
```c
typedef EFI_STATUS
                (EFIAPI *EFI_BLUETOOTH_IO_GET_DEVICE_INFO)(
                    IN EFI_BLUETOOTH_IO_PROTOCOL   *This,
                    OUT UINTN             *DeviceInfoSize,
                    OUT VOID             **DeviceInfo
                );
```

**Parameters**
- **This** Pointer to the **EFI_BLUETOOTH_IO_PROTOCOL** instance.
- **DeviceInfoSize** A pointer to the size, in bytes, of the **DeviceInfo** buffer.
- **DeviceInfo** A pointer to a callee allocated buffer that returns Bluetooth device information. Callee allocates this buffer by using EFI Boot Service **AllocatePool()**.

**Description**
The **GetDeviceInfo()** function returns Bluetooth device information. The size of **DeviceInfo** structure should never be assumed and the value of **DeviceInfoSize** is the only valid way to know the size of **DeviceInfo**.

**Related Definitions**
```c
typedef struct {
    UINT32        Version;
    BLUETOOTH_ADDRESS   BD_ADDR;
    UINT8          PageScanRepetitionMode;
    BLUETOOTH_CLASS_OF_DEVICE    ClassOfDevice;
    UINT16         ClockOffset;
    UINT8          RSSI;
    UINT8         ExtendedInquiryResponse[240];
} EFI_BLUETOOTH_DEVICE_INFO;
```

- **Version** The version of the structure. A value of zero represents the **EFI_BLUETOOTH_DEVICE_INFO** structure as defined here. Future version of this specification may extend this data structure in a backward compatible way and increase the value of **Version**.
- **BD_ADDR** 48bit Bluetooth device address.
PageScanRepetitionMode  
Bluetooth PageScanRepetitionMode. See Bluetooth specification for detail.

ClassOfDevice  
Bluetooth ClassOfDevice. See Bluetooth specification for detail.

ClockOffset  
Bluetooth ClockOffset. See Bluetooth specification for detail.

RSSI  
Bluetooth RSSI. See Bluetooth specification for detail.

ExtendedInquiryResponse  
Bluetooth ExtendedInquiryResponse. See Bluetooth specification for detail.

typedef struct {
    UINT8   Address[6];
} BLUETOOTH_ADDRESS;

typedef struct {
    UINT8   FormatType:2;
    UINT8   MinorDeviceClass: 6;
    UINT16  MajorDeviceClass: 5;
    UINT16  MajorServiceClass:11;
} BLUETOOTH_CLASS_OF_DEVICE;

Status Codes Returned

<table>
<thead>
<tr>
<th>Code</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>EFI_SUCCESS</td>
<td>The Bluetooth device information is returned successfully.</td>
</tr>
<tr>
<td>EFI_DEVICE_ERROR</td>
<td>A hardware error occurred trying to retrieve the Bluetooth device information.</td>
</tr>
</tbody>
</table>

**BLUETOOTH_IO_PROTOCOL.GetSdpInfo**

Summary
Get Bluetooth SDP information.

Prototype

typedef
EFI_STATUS
(EFIAPI *EFI_BLUETOOTH_IO_GET_SDP_INFO)(
    IN EFI_BLUETOOTH_IO_PROTOCOL   *This,
    OUT UINTN             *SdpInfoSize,
    OUT VOID             **SdpInfo
    );

Parameters

This  
Pointer to the EFI_BLUETOOTH_IO_PROTOCOL instance.

SdpInfoSize  
A pointer to the size, in bytes, of the SdpInfo buffer.
**SdpInfo**

A pointer to a callee allocated buffer that returns Bluetooth SDP information. Callee allocates this buffer by using EFI Boot Service `AllocatePool()`.

**Description**

The `GetSdpInfo()` function returns Bluetooth SDP information. The size of `SdpInfo` structure should never be assumed and the value of `SdpInfoSize` is the only valid way to know the size of `SdpInfo`.

**Status Codes Returned**

<table>
<thead>
<tr>
<th>Code</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>EFI_SUCCESS</td>
<td>The Bluetooth SDP information is returned successfully.</td>
</tr>
<tr>
<td>EFI_DEVICE_ERROR</td>
<td>A hardware error occurred trying to retrieve the Bluetooth SDP information.</td>
</tr>
</tbody>
</table>

**BLUETOOTH_IO_PROTOCOL.L2CapRawSend**

**Summary**

Send L2CAP message (including L2CAP header).

**Prototype**

```c
typedef EFI_STATUS (EFIAPI *EFI_BLUETOOTH_IO_L2CAP_RAW_SEND)(
    IN EFI_BLUETOOTH_IO_PROTOCOL   *This,
    IN OUT UINTN           *BufferSize,
    IN VOID              *Buffer,
    IN UINTN            Timeout
);
```

**Parameters**

- **This**
  Pointer to the `EFI_BLUETOOTH_IO_PROTOCOL` instance.
- **BufferSize**
  On input, indicates the size, in bytes, of the data buffer specified by `Buffer`. On output, indicates the amount of data actually transferred.
- **Buffer**
  A pointer to the buffer of data that will be transmitted to Bluetooth L2CAP layer.
- **Timeout**
  Indicating the transfer should be completed within this time frame. The units are in milliseconds. If `Timeout` is 0, then the caller must wait for the function to be completed until `EFI_SUCCESS` or `EFI_DEVICE_ERROR` is returned.

**Description**

The `L2CapRawSend()` function sends L2CAP layer message (including L2CAP header). `Buffer` holds the whole L2CAP message, including Length, ChannelID, and information payload. (see in Bluetooth specification, L2CAP Data Packet Format for more detail)
Status Codes Returned

<table>
<thead>
<tr>
<th>Status Code</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>EFI_SUCCESS</td>
<td>The L2CAP message is sent successfully.</td>
</tr>
<tr>
<td>EFI_INVALID_PARAMETER</td>
<td>One or more of the following conditions is TRUE:</td>
</tr>
<tr>
<td></td>
<td>- BufferSize is NULL.</td>
</tr>
<tr>
<td></td>
<td>- *BufferSize is 0.</td>
</tr>
<tr>
<td></td>
<td>- Buffer is NULL.</td>
</tr>
<tr>
<td>EFI_TIMEOUT</td>
<td>Sending L2CAP message fail due to timeout.</td>
</tr>
<tr>
<td>EFI_DEVICE_ERROR</td>
<td>Sending L2CAP message fail due to host controller or device error.</td>
</tr>
</tbody>
</table>

BLUETOOTH_IO_PROTOCOL.L2CapRawReceive

Summary
Receive L2CAP message (including L2CAP header).

Prototype
```c
typedef EFI_STATUS
  (EFIAPIC *EFI_BLUE TOOTH_IO_L2CAP_RAW_RECEIVE)(
    IN EFI_BLUE TOOTH_IO_PROTOCOL   *This,
    IN OUT UINTN           *BufferSize,
    OUT VOID             *Buffer,
    IN UINTN            Timeout
  );
```

Parameters
- **This**: Pointer to the EFI_BLUE TOOTH_IO_PROTOCOL instance.
- **BufferSize**: On input, indicates the size, in bytes, of the data buffer specified by Buffer. On output, indicates the amount of data actually transferred.
- **Buffer**: A pointer to the buffer of data that will be received from Bluetooth L2CAP layer.
- **Timeout**: Indicating the transfer should be completed within this time frame. The units are in milliseconds. If Timeout is 0, then the caller must wait for the function to be completed until EFI_SUCCESS or EFI_DEVICE_ERROR is returned.

Description
The L2CapRawReceive() function receives L2CAP layer message (including L2CAP header). Buffer holds the whole L2CAP message, including Length, ChannelID, and information payload. (see in Bluetooth specification, L2CAP Data Packet Format for more detail)
**Status Codes Returned**

<table>
<thead>
<tr>
<th>Code</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>EFI_SUCCESS</td>
<td>The L2CAP message is received successfully.</td>
</tr>
<tr>
<td>EFI_INVALID_PARAMETER</td>
<td>One or more of the following conditions is TRUE:</td>
</tr>
<tr>
<td></td>
<td>• BufferSize is NULL.</td>
</tr>
<tr>
<td></td>
<td>• *BufferSize is 0.</td>
</tr>
<tr>
<td></td>
<td>• Buffer is NULL.</td>
</tr>
<tr>
<td>EFI_TIMEOUT</td>
<td>Receiving L2CAP message fail due to timeout.</td>
</tr>
<tr>
<td>EFI_DEVICE_ERROR</td>
<td>Receiving L2CAP message fail due to host controller or device error.</td>
</tr>
</tbody>
</table>

**BLUETOOTH_IO_PROTOCOL.L2CapRawAsyncReceive**

**Summary**

Receive L2CAP message (including L2CAP header) in non-blocking way.

**Prototype**

```c
typedef
EFI_STATUS
(EFIAPI *EFI_BLUETOOTH_IO_L2CAP_RAW_ASYNC_RECEIVE)(
    IN EFI_BLUETOOTH_IO_PROTOCOL       *This,
    IN BOOLEAN                        IsNewTransfer,
    IN UINTN                          PollingInterval,
    IN UINTN                          DataLength,
    IN EFI_BLUETOOTH_IO_ASYNC_FUNC_CALLBACK  Callback,
    IN VOID                           *Context);
```

**Parameters**

- **This**
  Pointer to the `EFI_BLUETOOTH_IO_PROTOCOL` instance.

- **IsNewTransfer**
  If `TRUE`, a new transfer will be submitted. If `FALSE`, the request is deleted.

- **PollingInterval**
  Indicates the periodic rate, in milliseconds, that the transfer is to be executed.

- **DataLength**
  Specifies the length, in bytes, of the data to be received.

- **Callback**
  The callback function. This function is called if the asynchronous transfer is completed.

- **Context**
  Data passed into `Callback` function. This is optional parameter and may be `NULL`.

**Description**

The `L2CapRawAsyncReceive()` function receives L2CAP layer message (including L2CAP header) in non-blocking way. Data in `Callback` function holds the whole L2CAP message, including Length, ChannelID, and information payload. (see in Bluetooth specification, L2CAP Data Packet Format for more detail)
Related Definitions

typedef
   EFI_STATUS
(EFIAPI *EFI_BLUETOOTH_IO_ASYNC_FUNC_CALLBACK) (  
   IN UINT16 ChannelID,
   IN VOID *Data,
   IN UINTN DataLength,
   IN VOID *Context
);

ChannelID Bluetooth L2CAP message channel ID.
Data Data received via asynchronous transfer.
DataLength The length of Data in bytes, received via asynchronous transfer.
Context Context passed from asynchronous transfer request.

Status Codes Returned

<table>
<thead>
<tr>
<th>Code</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>EFI_SUCCESS</td>
<td>The L2CAP asynchronous receive request is submitted successfully.</td>
</tr>
<tr>
<td>EFI_INVALID_PARAMETER</td>
<td>One or more of the following conditions is TRUE:</td>
</tr>
<tr>
<td></td>
<td>• DataLength is 0.</td>
</tr>
<tr>
<td></td>
<td>• If IsNewTransfer is TRUE, and an asynchronous receive request already exists.</td>
</tr>
</tbody>
</table>

BLUETOOTH_IO_PROTOCOL.L2CapSend

Summary
Send L2CAP message (excluding L2CAP header) to a specific channel.

Prototype

typedef
   EFI_STATUS
(EFIAPI *EFI_BLUETOOTH_IO_L2CAP_SEND) (  
   IN EFI_BLUETOOTH_IO_PROTOCOL *This,
   IN EFI_HANDLE Handle,
   IN OUT UINTN *BufferSize,
   IN VOID *Buffer,
   IN UINTN Timeout
);

Parameters
This Pointer to the EFI_BLUETOOTH_IO_PROTOCOL instance.
Handle A handle created by EFI_BLUETOOTH_IO_PROTOCOL.L2CapConnect indicates which channel to send.
BufferSize
On input, indicates the size, in bytes, of the data buffer specified by Buffer. On output, indicates the amount of data actually transferred.

Buffer
A pointer to the buffer of data that will be transmitted to Bluetooth L2CAP layer.

Timeout
Indicating the transfer should be completed within this time frame. The units are in milliseconds. If Timeout is 0, then the caller must wait for the function to be completed until EFI_SUCCESS or EFI_DEVICE_ERROR is returned.

Description
The L2CapSend() function sends L2CAP layer message (excluding L2CAP header) to Bluetooth channel indicated by Handle. Buffer only holds information payload. (see in Bluetooth specification, L2CAP Data Packet Format for more detail). Handle

Status Codes Returned

<table>
<thead>
<tr>
<th>Code</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>EFI_SUCCESS</td>
<td>The L2CAP message is sent successfully.</td>
</tr>
<tr>
<td>EFI_NOT_FOUND</td>
<td>Handle is invalid or not found.</td>
</tr>
<tr>
<td>EFI_INVALID_PARAMETER</td>
<td>One or more of the following conditions is TRUE:</td>
</tr>
<tr>
<td></td>
<td>• BufferSize is NULL.</td>
</tr>
<tr>
<td></td>
<td>• *BufferSize is 0.</td>
</tr>
<tr>
<td></td>
<td>• Buffer is NULL.</td>
</tr>
<tr>
<td>EFI_TIMEOUT</td>
<td>Sending L2CAP message fail due to timeout.</td>
</tr>
<tr>
<td>EFI_DEVICE_ERROR</td>
<td>Sending L2CAP message fail due to host controller or device error.</td>
</tr>
</tbody>
</table>

BLUETOOTH_IO_PROTOCOL.L2CapReceive

Summary
Receive L2CAP message (excluding L2CAP header) from a specific channel.

Prototype

typedef

EFI_STATUS

(EIFIAPI *EFI_BLUETOOTH_IO_L2CAP_RECEIVE)(

  IN EFI_BLUETOOTH_IO_PROTOCOL  *This,

  IN EFI_HANDLE                 Handle,

  OUT UINTN                     *BufferSize,

  OUT VOID                      **Buffer,

  IN UINTN                      Timeout

);

Parameters

This Pointer to the EFI_BLUETOOTH_IO_PROTOCOL instance.
Handle

A handle created by EFI_BLUETOOTH_IO_PROTOCOL.L2CapConnect indicates which channel to receive.

BufferSize

Indicates the size, in bytes, of the data buffer specified by Buffer.

Buffer

A pointer to the buffer of data that will be received from Bluetooth L2CAP layer. Callee allocates this buffer by using EFI Boot Service AllocatePool().

Timeout

Indicating the transfer should be completed within this time frame. The units are in milliseconds. If Timeout is 0, then the caller must wait for the function to be completed until EFI_SUCCESS or EFI_DEVICE_ERROR is returned.

Description

The L2CapReceive() function receives L2CAP layer message (excluding L2CAP header) from Bluetooth channel indicated by Handle. Buffer only holds information payload. (see in Bluetooth specification, L2CAP Data Packet Format for more detail)

Status Codes Returned

<table>
<thead>
<tr>
<th>Status Code</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>EFI_SUCCESS</td>
<td>The L2CAP message is received successfully.</td>
</tr>
<tr>
<td>EFI_NOT_FOUND</td>
<td>Handle is invalid or not found.</td>
</tr>
<tr>
<td>EFI_INVALID_PARAMETER</td>
<td>One or more of the following conditions is TRUE:</td>
</tr>
<tr>
<td></td>
<td>• BufferSize is NULL.</td>
</tr>
<tr>
<td></td>
<td>• *BufferSize is 0.</td>
</tr>
<tr>
<td></td>
<td>• Buffer is NULL.</td>
</tr>
<tr>
<td>EFI_TIMEOUT</td>
<td>Receiving L2CAP message fail due to timeout.</td>
</tr>
<tr>
<td>EFI_DEVICE_ERROR</td>
<td>Receiving L2CAP message fail due to host controller or device error.</td>
</tr>
</tbody>
</table>

BLUETOOTH_IO_PROTOCOL.L2CapAsyncReceive

Summary

Receive L2CAP message (including L2CAP header) in non-blocking way from a specific channel.

Prototype

typedef

EFI_STATUS

(EIFIAPI *EFI_BLUETOOTH_IO_L2CAP_ASYNC_RECEIVE)(

  IN EFI_BLUETOOTH_IO_PROTOCOL *This,
  IN EFI_HANDLE Handle,
  IN EFI_BLUETOOTH_IO_CHANNEL_SERVICE_CALLBACK Callback,
  IN VOID *Context

Parameters

This Pointer to the EFI_BLUETOOTH_IO_PROTOCOL instance.
Handle
A handle created by
`EFI_BLUETOOTH_IO_PROTOCOL.L2CapConnect` indicates which channel to receive.

Callback
The callback function. This function is called if the asynchronous transfer is completed.

Context
Data passed into `Callback` function. This is optional parameter and may be `NULL`.

Description
The `L2CapAsyncReceive()` function receives L2CAP layer message (excluding L2CAP header) in non-blocking way from Bluetooth channel indicated by `Handle`. Data in `Callback` function only holds information payload. (see in Bluetooth specification, L2CAP Data Packet Format for more detail)

Related Definitions

```c
typedef EFI_STATUS (*EFI_BLUETOOTH_IO_CHANNEL_SERVICE_CALLBACK) (
    IN VOID *Data,
    IN UINTN DataLength,
    IN VOID *Context
);
```

- Data
  Data received via asynchronous transfer.
- DataLength
  The length of `Data` in bytes, received via asynchronous transfer.
- Context
  Context passed from asynchronous transfer request.

Status Codes Returned

<table>
<thead>
<tr>
<th>Status Code</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>EFI_SUCCESS</td>
<td>The L2CAP asynchronous receive request is submitted successfully.</td>
</tr>
<tr>
<td>EFI_NOT_FOUND</td>
<td><code>Handle</code> is invalid or not found.</td>
</tr>
<tr>
<td>EFI_INVALID_PARAMETER</td>
<td>One or more of the following conditions is TRUE:</td>
</tr>
<tr>
<td></td>
<td>• <code>DataLength</code> is 0.</td>
</tr>
<tr>
<td></td>
<td>• If an asynchronous receive request already exists on same <code>Handle</code>.</td>
</tr>
</tbody>
</table>

**BLUETOOTH_IO_PROTOCOL.L2CapConnect**

Summary
Do L2CAP connection.
Prototype

```c
typedef EFI_STATUS
(EIFIAPI *EFI_BLUETOOTH_IO_L2CAP_CONNECT)(
  IN EFI_BLUETOOTH_IO_PROTOCOL          *This,
  OUT EFI_HANDLE                 *Handle,
  IN UINT16                     Psm,
  IN UINT16                     Mtu,
  IN EFI_BLUETOOTH_IO_CHANNEL_SERVICE_CALLBACK  Callback,
  IN VOID                        *Context
);
```

Parameters

- **This**: Pointer to the `EFI_BLUETOOTH_IO_PROTOCOL` instance.
- **Handle**: A handle to indicate this L2CAP connection.
- **Psm**: Bluetooth PSM. See Bluetooth specification for detail.
- **Mtu**: Bluetooth MTU. See Bluetooth specification for detail.
- **Callback**: The callback function. This function is called whenever there is message received in this channel.
- **Context**: Data passed into `Callback` function. This is optional parameter and may be `NULL`.

Description

The `L2CapConnect()` function does all necessary steps for Bluetooth L2CAP layer connection in blocking way. It might take long time. Once this function is returned `Handle` is created to indicate the connection.

Status Codes Returned

<table>
<thead>
<tr>
<th>Code</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>EFI_SUCCESS</td>
<td>The Bluetooth L2CAP layer connection is created successfully.</td>
</tr>
<tr>
<td>EFI_INVALID_PARAMETER</td>
<td>One or more of the following conditions is TRUE:</td>
</tr>
<tr>
<td></td>
<td>• <code>Handle</code> is NULL.</td>
</tr>
<tr>
<td>EFI_DEVICE_ERROR</td>
<td>A hardware error occurred trying to do Bluetooth L2CAP connection.</td>
</tr>
</tbody>
</table>

**BLUETOOTH_IO_PROTOCOL.L2CapDisconnect**

Summary

Do L2CAP disconnection.
Prototype

typedef

EFI_STATUS

(EFIAPI *EFI_BLUETOOTH_IO_L2CAP_DISCONNECT)(
  IN EFI_BLUETOOTH_IO_PROTOCOL          *This,
  IN EFI_HANDLE                        Handle
);  

Parameters

This Pointer to the EFI_BLUETOOTH_IO_PROTOCOL instance.
Handle A handle to indicate this L2CAP connection.

Description

The L2CapDisconnect() function does all necessary steps for Bluetooth L2CAP layer disconnection in blocking way. It might take long time. Once this function is returned Handle is no longer valid.

Status Codes Returned

<table>
<thead>
<tr>
<th>Code</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>EFI_SUCCESS</td>
<td>The Bluetooth L2CAP layer disconnection is created successfully.</td>
</tr>
<tr>
<td>EFI_NOT_FOUND</td>
<td>Handle is invalid or not found.</td>
</tr>
<tr>
<td>EFI_DEVICE_ERROR</td>
<td>A hardware error occurred trying to do Bluetooth L2CAP disconnection.</td>
</tr>
</tbody>
</table>

BLUETOOTH_IO_PROTOCOL.L2CapRegisterService

Summary

Register L2CAP callback function for special channel.

Prototype

typedef

EFI_STATUS

(EFIAPI *EFI_BLUETOOTH_IO_L2CAP_REGISTER_SERVICE)(
  IN EFI_BLUETOOTH_IO_PROTOCOL          *This,
  OUT EFI_HANDLE                      *Handle,
  IN UINT16                           Psm,
  IN UINT16                           Mtu,
  IN EFI_BLUETOOTH_IO_CHANNEL_SERVICE_CALLBACK Callback,
  IN VOID                            *Context
);  

Parameters

This Pointer to the EFI_BLUETOOTH_IO_PROTOCOL instance.
Handle A handle to indicate this L2CAP connection.
Psm Bluetooth PSM. See Bluetooth specification for detail.
Mtut Bluetooth MTU. See Bluetooth specification for detail.
Callback

The callback function. This function is called whenever there is message received in this channel. **NULL** means unregister.

Context

Data passed into **Callback** function. This is optional parameter and may be **NULL**.

Description

The **L2CapRegisterService()** function registers L2CAP callback function for a special channel. Once this function is returned **Handle** is created to indicate the connection.

Status Codes Returned

<table>
<thead>
<tr>
<th>Code</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>EFI_SUCCESS</td>
<td>The Bluetooth L2CAP callback function is registered successfully.</td>
</tr>
<tr>
<td>EFI_ALREADY_STARTED</td>
<td>The callback function already exists when register.</td>
</tr>
<tr>
<td>EFI_NOT_FOUND</td>
<td>The callback function does not exist when unregister.</td>
</tr>
</tbody>
</table>

26.3 EFI Bluetooth Configuration Protocol

**EFI_BLUETOOTH_CONFIG_PROTOCOL**

Summary

This protocol abstracts user interface configuration for Bluetooth device.

GUID

```c
#define EFI_BLUETOOTH_CONFIG_PROTOCOL_GUID \
{ 0x62960cf3, 0x40ff, 0x4263,\ 
{ 0xa7, 0x7c, 0xdf, 0xde, 0xbd, 0x19, 0x1b, 0x4b }}
```

Protocol Interface Structure

```c
typedef struct _EFI_BLUETOOTH_CONFIG_PROTOCOL {
    EFI_BLUETOOTH_CONFIG_INIT Init;
    EFI_BLUETOOTH_CONFIG_SCAN Scan;
    EFI_BLUETOOTH_CONFIG_CONNECT Connect;
    EFI_BLUETOOTH_CONFIG_DISCONNECT Disconnect;
    EFI_BLUETOOTH_CONFIG_GET_DATA GetData;
    EFI_BLUETOOTH_CONFIG_SET_DATA GetData;
    EFI_BLUETOOTH_CONFIG_GET_REMOTE_DATA GetRemoteData;
    EFI_BLUETOOTH_CONFIG_REGISTER_PIN_CALLBACK RegisterPinCallback;
    EFI_BLUETOOTH_CONFIG_REGISTER_GET_LINK_KEY_CALLBACK RegisterGetLinkKeyCallback;
    EFI_BLUETOOTH_CONFIG_REGISTER_SET_LINK_KEY_CALLBACK RegisterSetLinkKeyCallback;
    EFI_BLUETOOTH_CONFIG_REGISTER_CONNECT_COMPLETE_CALLBACK RegisterLinkConnectCompleteCallback;
} EFI_BLUETOOTH_CONFIG_PROTOCOL;
```
Parameters

**Init**
Initialize Bluetooth host controller and local device. See the **Init()** function description.

**Scan**
Scan Bluetooth device. See the **Scan()** function description.

**Connect**
Connect one Bluetooth device. See the **Connect()** function description.

**Disconnect**
Disconnect one Bluetooth device. See the **Disconnect()** function description.

**GetData**
Get Bluetooth configuration data. See the **GetData()** function description.

**SetData**
Set Bluetooth configuration data. See the **SetData()** function description.

**GetRemoteData**
Get remote Bluetooth device data. See the **GetRemoteData()** function description.

**RegisterPinCallback**
Register PIN callback function. See the **RegisterPinCallback()** function description.

**RegisterGetLinkKeyCallback**
Register get link key callback function. See the **RegisterGetLinkKeyCallback()** function description.

**RegisterSetLinkKeyCallback**
Register set link key callback function. See the **RegisterSetLinkKeyCallback()** function description.

**RegisterLinkConnectCompleteCallback**
Register link connect complete callback function. See the **RegisterLinkConnectCompleteCallback()** function description.

Description

The **EFI_BLUETOOTH_CONFIG_PROTOCOL** abstracts the Bluetooth configuration. User can use Bluetooth configuration to interactive with Bluetooth bus driver.

**BLUETOOTH_CONFIG_PROTOCOL.Init**

Summary

Initialize Bluetooth host controller and local device.
Prototype

typedef

typedef

(EFI_STATUS)

EFI_SUCCESS

EFI_DEVICE_ERROR

The Bluetooth host controller and local device is initialized successfully.

A hardware error occurred trying to initialize the Bluetooth host controller and local device.

BLUETOOTH_CONFIG_PROTOCOL.Scan

Summary

Scan Bluetooth device.

Prototype

typedef

(EFI_STATUS)

IN EFI_BLUETOOTH_CONFIG_PROTOCOL *This,

IN BOOLEAN ReScan,

IN UINT8 ScanType,

IN EFI_BLUETOOTH_CONFIG_SCAN_CALLBACK_FUNCTION Callback

IN VOID *Context

);
Description

The `Scan()` function scans Bluetooth device. When this function is returned, it just means scan request is submitted. It does not mean scan process is started or finished. Whenever there is a Bluetooth device is found, the `Callback` function will be called. `Callback` function might be called before this function returns or after this function returns.

Related Definitions

```c
typedef EFI_STATUS
(EFIAPI *EFI_BLUETOOTH_CONFIG.ScanCallbackFunction)
(
    IN EFI_BLUETOOTH_CONFIG_PROTOCOL        *This,
    IN VOID                     *Context,
    IN EFI_BLUETOOTH_SCAN_CALLBACK_INFORMATION   *CallbackInfo
);
```

- **This** Pointer to the `EFI_BLUETOOTH_CONFIG_PROTOCOL` instance.
- **Context** Context passed from scan request.
- **CallbackInfo** Data related to scan result. `NULL CallbackInfo` means scan complete.

```c
typedef typedef struct{
    BLUETOOTH_ADDRESS BDAddr;
    UINT8 RemoteDeviceState;
    BLUETOOTH_CLASS_OF_DEVICE ClassOfDevice;
    UINT8 RemoteDeviceName[BLUETOOTH_HCI_COMMAND_LOCAL_READABLE_NAME_MAX_SIZE];
}EFI_BLUETOOTH_SCAN_CALLBACK_INFORMATION;
```

#define BLUETOOTH_HCI_COMMAND_LOCAL_READABLE_NAME_MAX_SIZE 248

Status Codes Returned

<table>
<thead>
<tr>
<th>Code</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>EFI_SUCCESS</td>
<td>The Bluetooth scan request is submitted.</td>
</tr>
<tr>
<td>EFI_DEVICE_ERROR</td>
<td>A hardware error occurred trying to scan the Bluetooth device.</td>
</tr>
</tbody>
</table>

**BLUETOOTH_CONFIG_PROTOCOL.Connect**

Summary

Connect a Bluetooth device.
Prototype

typedef
    EFI_STATUS
    (EFIAPI *EFI_BLUETOOTH_CONFIG_CONNECT)(
        IN EFI_BLUETOOTH_CONFIG_PROTOCOL *This,
        IN BLUETOOTH_ADDRESS *BD_ADDR
    );

Parameters

This                  Pointer to the EFI_BLUETOOTH_CONFIG_PROTOCOL instance.
BD_ADDR                The address of Bluetooth device to be connected.

Description

The Connect() function connects a Bluetooth device. When this function is returned successfully, a new EFI_BLUETOOTH_IO_PROTOCOL is created.

Status Codes Returned

<table>
<thead>
<tr>
<th>Status Code</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>EFI_SUCCESS</td>
<td>The Bluetooth device is connected successfully.</td>
</tr>
<tr>
<td>EFI_ALREADY_STARTED</td>
<td>The Bluetooth device is already connected.</td>
</tr>
<tr>
<td>EFI_NOT_FOUND</td>
<td>The Bluetooth device is not found.</td>
</tr>
<tr>
<td>EFI_DEVICE_ERROR</td>
<td>A hardware error occurred trying to connect the Bluetooth device.</td>
</tr>
</tbody>
</table>

BLUETOOTH_CONFIG_PROTOCOL.Disconnect

Summary

Disconnect a Bluetooth device.

Prototype

typedef
    EFI_STATUS
    (EFIAPI *EFI_BLUETOOTH_CONFIG_DISCONNECT)(
        IN EFI_BLUETOOTH_CONFIG_PROTOCOL *This,
        IN BLUETOOTH_ADDRESS *BD_ADDR,
        IN UINT8 Reason
    );

Parameters

This                  Pointer to the EFI_BLUETOOTH_CONFIG_PROTOCOL instance.
BD_ADDR                The address of Bluetooth device to be connected.
Reason                 Bluetooth disconnect reason. See Bluetooth specification for detail.
Description
The `Disconnect()` function disconnects a Bluetooth device. When this function is returned successfully, the `EFI_BLUETOOTH_IO_PROTOCOL` associated with this device is destroyed and all services associated are stopped.

Status Codes Returned

<table>
<thead>
<tr>
<th>Code</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>EFI_SUCCESS</td>
<td>The Bluetooth device is disconnected successfully.</td>
</tr>
<tr>
<td>EFI_NOT_STARTED</td>
<td>The Bluetooth device is not connected.</td>
</tr>
<tr>
<td>EFI_NOT_FOUND</td>
<td>The Bluetooth device is not found.</td>
</tr>
<tr>
<td>EFI_DEVICE_ERROR</td>
<td>A hardware error occurred trying to disconnect the Bluetooth device.</td>
</tr>
</tbody>
</table>

BLUETOOTH_CONFIG_PROTOCOL.GetData

Summary
Get Bluetooth configuration data.

Prototype
```c
typedef EFI_STATUS (EFIAPI *EFI_BLUETOOTH_CONFIG_GET_DATA) (
    IN EFI_BLUETOOTH_CONFIG_PROTOCOL    *This,
    IN EFI_BLUETOOTH_CONFIG_DATA_TYPE Data_Type,
    IN OUT UINTN              *DataSize,
    IN OUT VOID              *Data
);
```

Parameters
- **This**: Pointer to the `EFI_BLUETOOTH_CONFIG_PROTOCOL` instance.
- **DataType**: Configuration data type.
- **DataSize**: On input, indicates the size, in bytes, of the data buffer specified by `Data`. On output, indicates the amount of data actually returned.
- **Data**: A pointer to the buffer of data that will be returned.

Description
The `GetData()` function returns Bluetooth configuration data. For remote Bluetooth device configuration data, please use `GetRemoteData()` function with valid BD_ADDR.
Related Definitions

typedef enum {
    EfiBluetoothConfigDataTypeDeviceName,         /* Relevant for LE*/
    EfiBluetoothConfigDataTypeClassOfDevice,
    EfiBluetoothConfigDataTypeRemoteDeviceState,  /* Relevant for LE*/
    EfiBluetoothConfigDataTypeSdpInfo,
    EfiBluetoothConfigDataTypeBDADDR,            /* Relevant for LE*/
    EfiBluetoothConfigDataTypeDiscoverable,        /* Relevant for LE*/
    EfiBluetoothConfigDataTypeControllerStoredPairedDeviceList,
    EfiBluetoothConfigDataTypeAvailableDeviceList,
    EfiBluetoothConfigDataTypeRandomAddress,    /* Relevant for LE*/
    EfiBluetoothConfigDataTypeRSSI,                /* Relevant for LE*/
    EfiBluetoothConfigDataTypeAdvertisementData,  /* Relevant for LE*/
    EfiBluetoothConfigDataTypeIoCapability,         /* Relevant for LE*/
    EfiBluetoothConfigDataTypeOOBDataFlag,       /* Relevant for LE*/
    EfiBluetoothConfigDataTypeKeyType,           /* Relevant for LE*/
    EfiBluetoothConfigDataTypeEncKeySize,         /* Relevant for LE*/
    EfiBluetoothConfigDataTypeMax,
} EFI_BLUETOOTH_CONFIG_DATA_TYPE;

EfiBluetoothConfigDataTypeAdvertisementDataReport
    Advertisement report. Data structure is UINT8[].

EfiBluetoothConfigDataTypeKeyType
    KeyType of Authentication Requirements flag of local device
    as UINT8, indicating requested security properties. See Bluetooth
    specification 3.H.3.5.1. BIT0: MITM, BIT1: SC.

EfiBluetoothConfigDataTypeDeviceName
    Local/Remote Bluetooth device name. Data structure is zero
    terminated CHAR8[].

EfiBluetoothConfigDataTypeClassOfDevice
    Local/Remote Bluetooth device ClassOfDevice. Data structure is
    BLUETOOTH_CLASS_OF_DEVICE.

EfiBluetoothConfigDataTypeRemoteDeviceState
    Remote Bluetooth device state. Data structure is
    EFI_BLUETOOTH_CONFIG_REMOTE_DEVICE_STATE_TYPE.

EfiBluetoothConfigDataTypeSdpInfo
    Local/Remote Bluetooth device SDP information. Data structure is
    UINT8[].

EfiBluetoothConfigDataTypeBDADDR
    Local Bluetooth device address. Data structure is
    BLUETOOTH_ADDRESS.

EfiBluetoothConfigDataTypeDiscoverable
Local Bluetooth discoverable state. Data structure is `UINT8`. (Page scan and/or Inquiry scan)

```c
EfiBluetoothConfigDataTypeControllerStoredPairedDeviceList
```

Local Bluetooth controller stored paired device list. Data structure is `BLUETOOTH_ADDRESS[]`.

```c
EfiBluetoothConfigDataTypeAvailableDeviceList
```

Local available device list. Data structure is `BLUETOOTH_ADDRESS[]`.

```c
typedef EFI_BLUETOOTH_CONFIG_REMOTE_DEVICE_STATE_TYPE UINT32;
#define EFI_BLUETOOTH_CONFIG_REMOTE_DEVICE_STATE_CONNECTED 0x1
#define EFI_BLUETOOTH_CONFIG_REMOTE_DEVICE_STATE_PAIRED   0x2

#define BLUETOOTH_HCI_LINK_KEY_SIZE 16
```

**Status Codes Returned**

<table>
<thead>
<tr>
<th>Status Code</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>EFI_SUCCESS</td>
<td>The Bluetooth configuration data is returned successfully.</td>
</tr>
<tr>
<td>EFI_INVALID_PARAMETER</td>
<td>One or more of the following conditions is TRUE:</td>
</tr>
<tr>
<td></td>
<td>• DataSize is NULL.</td>
</tr>
<tr>
<td></td>
<td>• *DataSize is not 0 and Data is NULL</td>
</tr>
<tr>
<td>EFI_UNSUPPORTED</td>
<td>The DataType is unsupported.</td>
</tr>
<tr>
<td>EFI_NOT_FOUND</td>
<td>The DataType is not found.</td>
</tr>
<tr>
<td>EFI_BUFFER_TOO_SMALL</td>
<td>The buffer is too small to hold the buffer.</td>
</tr>
<tr>
<td></td>
<td>*DataSize has been updated with the size needed to complete the request.</td>
</tr>
</tbody>
</table>

**BLUETOOTH_CONFIG_PROTOCOL.SetData**

**Summary**

Set Bluetooth configuration data.

**Prototype**

```c
typedef
EFI_STATUS
(EIFIAPI *EFI_BLUETOOTH_CONFIG_SET_DATA) (  
    IN EFI_BLUETOOTH_CONFIG_PROTOCOL    *This,  
    IN EFI_BLUETOOTH_CONFIG_DATA_TYPE   DataType,  
    IN UINTN                           DataSize,  
    IN VOID                            *Data  
);
```
Parameters

This
DataType
DataSize
Data

Pointer to the `EFI_BLUETOOTH_CONFIG_PROTOCOL` instance.
Configuration data type.
Indicates the size, in bytes, of the data buffer specified by `Data`.
A pointer to the buffer of data that will be set.

Description

The `SetData()` function sets local Bluetooth device configuration data. Not all `DataType` can be set.

Status Codes Returned

<table>
<thead>
<tr>
<th>Code</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>EFI_SUCCESS</td>
<td>The Bluetooth configuration data is set successfully.</td>
</tr>
<tr>
<td>EFI_INVALID_PARAMETER</td>
<td>One or more of the following conditions is TRUE:</td>
</tr>
<tr>
<td></td>
<td>• <code>DataSize</code> is 0.</td>
</tr>
<tr>
<td></td>
<td>• <code>Data</code> is NULL.</td>
</tr>
<tr>
<td>EFI_UNSUPPORTED</td>
<td>The <code>DataType</code> is unsupported.</td>
</tr>
<tr>
<td>EFI_WRITE_PROTECTED</td>
<td>Cannot set configuration data.</td>
</tr>
</tbody>
</table>

**BLUETOOTH_CONFIG_PROTOCOL.GetRemoteData**

Summary

Get remove Bluetooth device configuration data.

Prototype

```c
typedef
EFI_STATUS
(EFIAPI *EFI_BLUETOOTH_CONFIG_GET_REMOTE_DATA) (    
    IN EFI_BLUETOOTH_CONFIG_PROTOCOL    *This,    
    IN EFI_BLUETOOTH_CONFIG_DATA_TYPE   DataType,    
    IN BLUETOOTH_ADDRESS               *BDAddr,    
    IN OUT UINTN                       *DataSize,    
    IN OUT VOID                        *Data    
);    
```

Parameters

This
DataType
BDAddr
DataSize
Data

Pointer to the `EFI_BLUETOOTH_CONFIG_PROTOCOL` instance.
Configuration data type.
Remote Bluetooth device address.
On input, indicates the size, in bytes, of the data buffer specified by `Data`. On output, indicates the amount of data actually returned.
A pointer to the buffer of data that will be returned.

Description

The `GetRemoteData()` function returns remote Bluetooth device configuration data.
Status Codes Returned

<table>
<thead>
<tr>
<th>Code</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>EFI_SUCCESS</td>
<td>The remote Bluetooth device configuration data is returned successfully.</td>
</tr>
<tr>
<td>EFI_INVALID_PARAMETER</td>
<td>One or more of the following conditions is TRUE:</td>
</tr>
<tr>
<td></td>
<td>• DataSize is NULL.</td>
</tr>
<tr>
<td></td>
<td>• *DataSize is not 0 and Data is NULL</td>
</tr>
<tr>
<td>EFI_UNSUPPORTED</td>
<td>The DataType is unsupported.</td>
</tr>
<tr>
<td>EFI_NOT_FOUND</td>
<td>The DataType is not found.</td>
</tr>
<tr>
<td>EFI_BUFFER_TOO_SMALL</td>
<td>The buffer is too small to hold the buffer.</td>
</tr>
<tr>
<td></td>
<td>*DataSize has been updated with the size needed to complete the request.</td>
</tr>
</tbody>
</table>

BLUETOOTH_CONFIG_PROTOCOL.RegisterPinCallback

Summary
Register PIN callback function.

Prototype
```c
typedef EFI_STATUS
  (EFIAPI *EFI_BLUETOOTH_CONFIG_REGISTER_PIN_CALLBACK) (
   IN EFI_BLUETOOTH_CONFIG_PROTOCOL            *This,
   IN EFI_BLUETOOTH_CONFIG_REGISTER_PIN_CALLBACK_FUNCTION Callback,
   IN VOID                         *Context
);
```

Parameters
- This: Pointer to the EFI_BLUETOOTH_CONFIG_PROTOCOL instance.
- Callback: The callback function. NULL means unregister.
- Context: Data passed into Callback function. This is optional parameter and may be NULL.

Description
The RegisterPinCallback() function registers Bluetooth PIN callback function. The Bluetooth configuration driver must call RegisterPinCallback() to register a callback function. During pairing, Bluetooth bus driver must trigger this callback function, and Bluetooth configuration driver must handle callback function according to CallbackType during pairing. Both Legacy pairing and SSP (secure simple pairing) are required to be supported. See EFI_BLUETOOTH_PIN_CALLBACK_TYPE below for detail of each pairing mode.
Related Definitions

typedef

EFI_STATUS

(EIFIAPI *EFI_BLUETOOTH_CONFIG_REGISTER_PIN_CALLBACK_FUNCTION) (  
  IN EFI_BLUETOOTH_CONFIG_PROTOCOL  *This,  
  IN VOID                *Context,  
  IN EFI_BLUETOOTH_PIN_CALLBACK_TYPE   CallbackType,  
  IN VOID                *InputBuffer,  
  IN UINTN                  InputBufferSize,  
  OUT VOID                **OutputBuffer,  
  OUT UINTN               *OutputBufferSize
);

This Pointer to the EFI_BLUETOOTH_CONFIG_PROTOCOL instance.
Context Context passed from registration.
CallbackType Callback type in EFI_BLUETOOTH_PIN_CALLBACK_TYPE.
InBuffer A pointer to the buffer of data that is input from callback caller.
InputBufferSize Indicates the size, in bytes, of the data buffer specified by InBuffer.
OutputBuffer A pointer to the buffer of data that will be output from callback callee. Callee allocates this buffer by using EFI Boot Service AllocatePool().
OutputBufferSize Indicates the size, in bytes, of the data buffer specified by OutputBuffer.

typedef enum {
  EfiBluetoothCallbackTypeUserPasskeyNotification,  
  EfiBluetoothCallbackTypeUserConfirmationRequest,  
  EfiBluetoothCallbackTypeOOBDataRequest,  
  EfiBluetoothCallbackTypePinCodeRequest,  
  EfiBluetoothCallbackTypeMax,  
  EFI_BLUETOOTH_PIN_CALLBACK_TYPE;

  EfiBluetoothCallbackTypeUserPasskeyNotification
    For SSP – passkey entry. Input buffer is Passkey (4 bytes). No output buffer. See Bluetooth HCI command for detail.

  EfiBluetoothCallbackTypeUserConfirmationRequest
    For SSP – just work and numeric comparison. Input buffer is numeric value (4 bytes). Output buffer is BOOLEAN (1 byte). See Bluetooth HCI command for detail.

  EfiBluetoothCallbackTypeOOBDataRequest
    For SSP – OOB. See Bluetooth HCI command for detail.

  EfiBluetoothCallbackTypePinCodeRequest
    For legacy paring. No input buffer. Output buffer is PIN code (<= 16 bytes). See Bluetooth HCI command for detail.
Status Codes Returned

<table>
<thead>
<tr>
<th>Code</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>EFI_SUCCESS</td>
<td>The PIN callback function is registered successfully.</td>
</tr>
</tbody>
</table>

**BLUETOOTH_CONFIG_PROTOCOL.RegisterGetLinkKeyCallback**

**Summary**

Register get link key callback function.

**Prototype**

```c
typedef EFI_STATUS (EFIAPI *EFI_BLUETOOTH_CONFIG_REGISTER_GET_LINK_KEY_CALLBACK) (
    IN EFI_BLUETOOTH_CONFIG_PROTOCOL *This,
    IN EFI_BLUETOOTH_CONFIG_REGISTER_GET_LINK_KEY_CALLBACK_FUNCTION Callback,
    IN VOID *Context)
```

**Parameters**

- **This**: Pointer to the `EFI_BLUETOOTH_CONFIG_PROTOCOL` instance.
- **Callback**: The callback function. **NULL** means unregister.
- **Context**: Data passed into `Callback` function. This is optional parameter and may be **NULL**.

**Description**

The `RegisterGetLinkKeyCallback()` function registers Bluetooth get link key callback function. The Bluetooth configuration driver may call `RegisterGetLinkKeyCallback()` to register a callback function. When Bluetooth bus driver get Link_Key_Request_Event, Bluetooth bus driver must trigger this callback function if it is registered. Then the callback function in Bluetooth configuration driver must pass link key to Bluetooth bus driver. When the callback function is returned Bluetooth bus driver gets link key and must send HCI_Link_Key_Request_Reply to remote device. If this `GetLinkKey` callback function is not registered or Bluetooth configuration driver fails to return a valid link key, the Bluetooth bus driver must send HCI_Link_Key_Request_Negative_Reply to remote device. The original link key is passed by Bluetooth bus driver to Bluetooth configuration driver by using `EFI_BLUETOOTH_CONFIG_REGISTER_SET_LINK_KEY_CALLBACK_FUNCTION`. The Bluetooth configuration driver need save link key to a non-volatile safe place. (See Bluetooth specification, HCI_Link_Key_Request_Reply)
Related Definitions

```c
typedef
EFI_STATUS
(EFIAPI *EFI_BLUETOOTH_CONFIG_REGISTER_GET_LINK_KEY_CALLBACK_FUNCTION) (
    IN EFI_BLUETOOTH_CONFIG_PROTOCOL   *This,
    IN VOID                *Context,
    IN BLUETOOTH_ADDRESS         *BDAddr,
    OUT UINT8               LinkKey[BLUETOOTH_HCI_LINK_KEY_SIZE]
);
```
This Pointer to the `EFI_BLUETOOTH_CONFIG_PROTOCOL` instance.
Context Context passed from registration.
CallbackType Callback type in `EFI_BLUETOOTH_PIN_CALLBACK_TYPE`.
BDAddr A pointer to Bluetooth device address.
LinkKey A pointer to the buffer of link key.

Status Codes Returned

<table>
<thead>
<tr>
<th>Status Code</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>EFI_SUCCESS</td>
<td>The link key callback function is registered successfully.</td>
</tr>
</tbody>
</table>

**BLUETOOTH_CONFIG_PROTOCOL/RegisterSetLinkKeyCallback**

**Summary**
Register set link key callback function.

**Prototype**

```c
typedef
EFI_STATUS
(EFIAPI *EFI_BLUETOOTH_CONFIG_REGISTER_SET_LINK_KEY_CALLBACK) (
    IN EFI_BLUETOOTH_CONFIG_PROTOCOL                 *This,
    IN EFI_BLUETOOTH_CONFIG_REGISTER_SET_LINK_KEY_CALLBACK_FUNCTION
    Callback,
    IN VOID                              *Context
);
```

**Parameters**

- **This** Pointer to the `EFI_BLUETOOTH_CONFIG_PROTOCOL` instance.
- **Callback** The callback function. `NULL` means unregister.
- **Context** Data passed into `Callback` function. This is optional parameter and may be `NULL`.

**Description**
The `RegisterSetLinkKeyCallback()` function registers Bluetooth link key callback function. The Bluetooth configuration driver may call `RegisterSetLinkKeyCallback()` to register a callback function to get link key from Bluetooth bus driver. When Bluetooth bus driver gets Link_Key_Notification_Event, Bluetooth bus driver must call this callback function if it is registered. Then
the callback function in Bluetooth configuration driver must save link key to a safe place. This link key will be used by `EFI_BLUETOOTH_CONFIG_REGISTER_GET_LINK_KEY_CALLBACK_FUNCTION` later. (See Bluetooth specification, Link_Key_Notification_Event)

**Related Definitions**

typedef

```c
EFI_STATUS
(EFIAPI *EFI_BLUETOOTH_CONFIG_REGISTER_SET_LINK_KEY_CALLBACK_FUNCTION) ( 
    IN EFI_BLUETOOTH_CONFIG_PROTOCOL   *This,
    IN VOID                *Context,
    IN BLUETOOTH_ADDRESS         *BDAddr,
    IN UINT8               LinkKey[BLUETOOTH_HCI_LINK_KEY_SIZE]
);
```

- **This**: Pointer to the `EFI_BLUETOOTH_CONFIG_PROTOCOL` instance.
- **Context**: Context passed from registration.
- **CallbackType**: Callback type in `EFI_BLUETOOTH_PIN_CALLBACK_TYPE`.
- **BDAddr**: A pointer to Bluetooth device address.
- **LinkKey**: A pointer to the buffer of link key.

**Status Codes Returned**

<table>
<thead>
<tr>
<th>Code</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>EFI_SUCCESS</td>
<td>The link key callback function is registered successfully.</td>
</tr>
</tbody>
</table>

**BLUETOOTH_CONFIG_PROTOCOL.RegisterLinkConnectCompleteCallback**

**Summary**

Register link connect complete callback function.

**Prototype**

typedef

```c
EFI_STATUS
(EFIAPI *EFI_BLUETOOTH_CONFIG_REGISTER_CONNECT_COMPLETE_CALLBACK) ( 
    IN EFI_BLUETOOTH_CONFIG_PROTOCOL                    *This,
    IN EFI_BLUETOOTH_CONFIG_REGISTER_CONNECT_COMPLETE_CALLBACK_FUNCTION
    Callback,
    IN VOID                                 *Context
);
```

- **This**: Pointer to the `EFI_BLUETOOTH_CONFIG_PROTOCOL` instance.
- **Callback**: The callback function. **NULL** means unregister.
- **Context**: Data passed into `Callback` function. This is optional parameter and may be **NULL**.
**Description**

The `RegisterLinkConnectCompleteCallback()` function registers Bluetooth link connect complete callback function. The Bluetooth Configuration driver may call `RegisterLinkConnectCompleteCallback()` to register a callback function. During pairing, Bluetooth bus driver must trigger this callback function to report device state, if it is registered. Then Bluetooth Configuration driver will get information on device connection, according to `CallbackType` defined by `EFI_BLUETOOTH_CONNECT_COMPLETE_CALLBACK_TYPE`.

**Related Definitions**

```c
typedef EFI_STATUS (EFIAPI *EFI_BLUETOOTH_CONFIG_REGISTER_CONNECT_COMPLETE_CALLBACK_FUNCTION) (  
    IN EFI_BLUETOOTH_CONFIG_PROTOCOL *This,  
    IN VOID *Context,  
    IN EFI_BLUETOOTH_CONNECT_COMPLETE_CALLBACK_TYPE CallbackType,  
    IN BLUETOOTH_ADDRESS *BDAddr,  
    IN VOID *InputBuffer,  
    IN UINTN InputBufferSize);  
```

- **This** Pointer to the `EFI_BLUETOOTH_CONFIG_PROTOCOL` instance.
- **Context** Context passed from registration.
- **CallbackType** Callback type in `EFI_BLUETOOTH_CONNECT_COMPLETE_CALLBACK_TYPE`.
- **BDAddr** A pointer to Bluetooth device address.
- **InputBuffer** A pointer to the buffer of data that is input from callback caller.
- **InputBufferSize** Indicates the size, in bytes, of the data buffer specified by `InputBuffer`.

```c
typedef enum {  
    EfiBluetoothConnCallbackTypeDisconnected,  
    EfiBluetoothConnCallbackTypeConnected,  
    EfiBluetoothConnCallbackTypeAuthenticated,  
    EfiBluetoothConnCallbackTypeEncrypted,  
} EFI_BLUETOOTH_CONNECT_COMPLETE_CALLBACK_TYPE;
```

- **EfiBluetoothConnCallbackTypeDisconnected** This callback is called when Bluetooth receive Disconnection_Complete event. Input buffer is Event Parameters of Disconnection_Complete Event defined in Bluetooth specification.

- **EfiBluetoothConnCallbackTypeConnected** This callback is called when Bluetooth receive Connection_Complete event. Input buffer is Event Parameters of Connection_Complete Event defined in Bluetooth specification.

- **EfiBluetoothConnCallbackTypeAuthenticated**
This callback is called when Bluetooth receive Authentication_Complete event. Input buffer is Event Parameters of Authentication_Complete Event defined in Bluetooth specification.

EfiBluetoothConnCallbackTypeEncrypted

This callback is called when Bluetooth receive Encryption_Change event. Input buffer is Event Parameters of Encryption_Change Event defined in Bluetooth specification.

Status Codes Returned

<table>
<thead>
<tr>
<th>Status Code</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>EFI_SUCCESS</td>
<td>The link connect complete callback function is registered successfully.</td>
</tr>
</tbody>
</table>

26.4 EFI Bluetooth Attribute Protocol

EFI_BLUETOOTH_ATTRIBUTE_PROTOCOL

Summary
This protocol provides service for Bluetooth ATT (Attribute Protocol) and GATT (Generic Attribute Profile) based protocol interfaces.

GUID
#define EFI_BLUETOOTH_ATTRIBUTE_PROTOCOL_GUID \  
  { 0x898890e9, 0x84b2, 0x4f3a, \{ 0x8c, 0x58, 0xd8, 0x57, 0x78, 0x13, 0xe0, \} 0xac }

Protocol Interface Structure
typedef struct _EFI_BLUETOOTH_ATTRIBUTE_PROTOCOL {  
  EFI_BLUETOOTH_ATTRIBUTE_SEND_REQUEST SendRequest; \  
  EFI_BLUETOOTH_ATTRIBUTE_REGISTER_FOR_SERVER_NOTIFICATION RegisterForServerNotification; \  
  EFI_BLUETOOTH_ATTRIBUTE_GET_SERVICE_INFO GetServiceInfo; \  
  EFI_BLUETOOTH_ATTRIBUTE_GET_DEVICE_INFO GetDeviceInfo; \  
} EFI_BLUETOOTH_ATTRIBUTE_PROTOCOL;

Parameters

SendRequest  Send a “REQUEST” or “COMMAND” message to remote server and receive a “RESPONSE” message for “REQUEST” from remote server according to Bluetooth attribute protocol data unit (PDU). See the SendRequest() function description.

RegisterForServerNotification  Register or unregister a server initiated PDU, such as “NOTIFICATION” or “INDICATION” on a characteristic value on remote server. See the RegisterForServerInitiatedMessage() function description.

GetServiceInfo  Get discovered service data information from connected remote device. See GetServiceInfo() function description.
GetDeviceInfo

Get the device information. See \texttt{GetDeviceInfo()} function description.

\textbf{Description}

The \texttt{EFI\_BLUETOOTH\_ATTRIBUTE\_PROTOCOL} provides services in ATT protocol and GATT profile. For detail of ATT protocol, and GATT profile, please refer to Bluetooth specification.

\textbf{BLUETOOTH\_ATTRIBUTE\_PROTOCOL.SendRequest}

\textbf{Summary}

Send a “REQUEST” or “COMMAND” message to remote server and receive a “RESPONSE” message for “REQUEST” from remote server according to Bluetooth attribute protocol data unit (PDU).

\textbf{Prototype}

\begin{verbatim}
typedef EFI_STATUS
  (EFIAPI *EFI_BLUETOOTH_ATTRIBUTE_SEND_REQUEST)(
    IN EFI_BLUETOOTH_ATTRIBUTE_PROTOCOL  *This,
    IN VOID  Data,
    IN UINTN  DataLength,
    IN EFI_BLUETOOTH_ATTRIBUTE_CALLBACK_FUNCTION  Callback,
    IN VOID  Context
  );
\end{verbatim}

\textbf{Parameters}

\begin{itemize}
  \item \texttt{This} Pointer to the EFI\_BLUETOOTH\_ATTRIBUTE\_PROTOCOL instance.
  \item \texttt{Data} Data of a REQUEST or COMMAND message. The first byte is the attribute PDU related opcode, followed by opcode specific fields. See Bluetooth specification, Vol 3, Part F, Attribute Protocol.
  \item \texttt{DataLength} The length of \texttt{Data} in bytes.
  \item \texttt{Callback} Callback function to notify the RESPONSE is received to the caller, with the response buffer. Caller must check the response buffer content to know if the request action is success or fail. It may be NULL if the data is a COMMAND.
  \item \texttt{Context} Data passed into \texttt{Callback} function. It is optional parameter and may be NULL.
\end{itemize}
Description

The `SendRequest()` function sends a “REQUEST” or “COMMAND” message to remote server and receive a “RESPONSE” message for “REQUEST” from remote server according to Bluetooth attribute protocol data unit (PDU). In most cases, this interface is used to read attributes from remote device, or write attributes to remote device.

Related Definitions

typedef

```c
EFI_STATUS (EFIAPI *EFI_BLUETOOTH_ATTRIBUTE_CALLBACK_FUNCTION) (
    IN EFI_BLUETOOTH_ATTRIBUTE_PROTOCOL *This,
    IN VOID *Data,
    IN UINTN DataLength,
    IN VOID *Context
);
```

- **This**: Pointer to the EFI_BLUETOOTH_ATTRIBUTE_PROTOCOL instance.
- **Data**: Data received. The first byte is the attribute opcode, followed by opcode specific fields. See Bluetooth specification, Vol 3, Part F, Attribute Protocol. It might be a normal RESPONSE message, or ERROR RESPONSE message.
- **DataLength**: The length of `Data` in bytes.
- **Context**: The context passed from the callback registration request.

Status Codes Returned

<table>
<thead>
<tr>
<th>Code</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>EFI_SUCCESS</td>
<td>The request is sent successfully.</td>
</tr>
<tr>
<td>EFI_INVALID_PARAMETER</td>
<td>One or more parameters are invalid due to following conditions:</td>
</tr>
<tr>
<td></td>
<td>- The <code>Buffer</code> is NULL.</td>
</tr>
<tr>
<td></td>
<td>- The <code>BufferLength</code> is 0.</td>
</tr>
<tr>
<td></td>
<td>- The opcode in <code>Buffer</code> is not a valid OPCODE according to Bluetooth</td>
</tr>
<tr>
<td></td>
<td>specification.</td>
</tr>
<tr>
<td></td>
<td>- The <code>Callback</code> is NULL.</td>
</tr>
<tr>
<td>EFI_DEVICE_ERROR</td>
<td>Sending the request failed due to the host controller or the device error.</td>
</tr>
<tr>
<td>EFI_NOT_READY</td>
<td>A GATT operation is already underway for this device</td>
</tr>
<tr>
<td>EFI_UNSUPPORTED</td>
<td>The attribute does not support the corresponding operation</td>
</tr>
</tbody>
</table>

**BLUETOOTH_ATTRIBUTE_PROTOCOL.RegisterForServerNotification**

Summary

Register or unregister a server initiated message, such as NOTIFICATION or INDICATION, on a characteristic value on remote server.

Prototype
typedef

EFI_STATUS

(EIFI_API *EFI_BLUETOOTH_ATTRIBUTE_REGISTER_FOR_SERVER_NOTIFICATION)(
    IN EFI_BLUETOOTH_ATTRIBUTE_PROTOCOL *This,
    IN EFI_BLUETOOTH_ATTRIBUTE_CALLBACK_PARAMETER *CallbackParameter,
    IN EFI_BLUETOOTH_ATTRIBUTE_CALLBACK_FUNCTION Callback,
    IN VOID *Context
);

Parameters

This              Pointer to the EFI_BLUETOOTH_ATTRIBUTE_PROTOCOL instance.

CallbackParameterThe parameter of the callback.

Callback         Callback function for server initiated attribute protocol. NULL callback function means unregister the server initiated callback.

Context          Data passed into Callback function. It is optional parameter and may be NULL.

Description

The RegisterForServerNotification() function can be issued to request Bluetooth to register or unregister a server initiated message, such as notification or indication, on a characteristic value on remote server. It can only be done if the characteristic supports that operation.

Related Definitions

typedef struct {
    UINT16   AttributeHandle;
} EFI_BLUETOOTH_ATTRIBUTE_CALLBACK_PARAMETER_NOTIFICATION;
typedef struct {
    UINT16   AttributeHandle;
} EFI_BLUETOOTH_ATTRIBUTE_CALLBACK_PARAMETER_INDICATION;
typedef struct {
    UINT32   Version;
    UINT8    AttributeOpCode;

    union {
        EFI_BLUETOOTH_ATTRIBUTE_CALLBACK_PARAMETER_NOTIFICATION Notification;
        EFI_BLUETOOTH_ATTRIBUTE_CALLBACK_PARAMETER_INDICATION   Indication;
    } Parameter;
} EFI_BLUETOOTH_ATTRIBUTE_CALLBACK_PARAMETER;
Version

The version of the structure. A value of zero represents the `EFI_BLUETOOTH_ATTRIBUTE_CALLBACK_PARAMETER` structure as defined here. Future version of this specification may extend this data structure in a backward compatible way and increase the value of `Version`.

AttributeOpCode


AttributeHandle

The attribute handle for notification or indication.

Status Codes Returned

<table>
<thead>
<tr>
<th>Code</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>EFI_SUCCESS</td>
<td>The callback function is registered or unregistered successfully</td>
</tr>
<tr>
<td>EFI_INVALID_PARAMETER</td>
<td>The attribute opcode is not server initiated message opcode. See Bluetooth specification, Vol 3, Part F, Attribute Protocol.</td>
</tr>
<tr>
<td>EFI_ALREADY_STARTED</td>
<td>A callback function is already registered on the same attribute opcode and attribute handle, when the <code>Callback</code> is not NULL.</td>
</tr>
<tr>
<td>EFI_NOT_STARTED</td>
<td>A callback function is not registered on the same attribute opcode and attribute handle, when the <code>Callback</code> is NULL.</td>
</tr>
<tr>
<td>EFI_NOT_READY</td>
<td>A GATT operation is already underway for this device</td>
</tr>
<tr>
<td>EFI_UNSUPPORTED</td>
<td>The attribute does not support notification</td>
</tr>
</tbody>
</table>

**BLUETOOTH_ATTRIBUTE_PROTOCOL.GetServiceInfo**

**Summary**

Get Bluetooth discovered service information.

**Prototype**

```c
typedef EFI_STATUS
  (EFIAPI *EFI_BLUETOOTH_ATTRIBUTE_GET_SERVICE_INFO)(
   IN EFI_BLUETOOTH_ATTRIBUTE_PROTOCOL *This,
   OUT UINTN *ServiceInfoSize,
   OUT VOID **ServiceInfo
  );
```

**Parameters**

- **This**
  
  Pointer to the `EFI_BLUETOOTH_ATTRIBUTE_PROTOCOL` instance.

- **ServiceInfoSize**
  
  A pointer to the size, in bytes, of the `ServiceInfo` buffer.

- **ServiceInfo**
  
  A pointer to a callee allocated buffer that returns Bluetooth discovered service information. Callee allocates this buffer by using EFI Boot Service `AllocatePool()`.

**Description**

The `GetServiceInfo()` function returns Bluetooth discovered service information. The size of `ServiceInfo` structure should never be assumed and the value of `ServiceInfoSize` is the only valid way to know the size of `ServiceInfo`. The
ServiceInfo buffer is a list Bluetooth service information structures defined below.

Related Definitions

```c
typedef struct {
    UINT8         Length;
    union {
        UINT16       Uuid16;
        UINT32       Uuid32;
        UINT8        Uuid128[16];
    } Data;
} EFI_BLUETOOTH_UUID;
```

- **Length**: The length of Bluetooth UUID data. The valid value is 2, 4, or 16.
- **Uuid16**: The 16-bit Bluetooth UUID data.
- **Uuid32**: The 32-bit Bluetooth UUID data.
- **Uuid128**: The 128-bit Bluetooth UUID data.

```c
typedef struct {
    EFI_BLUETOOTH_UUID         Type;
    UINT16               Length;
    UINT16               AttributeHandle;
    EFI_BLUETOOTH_ATTRIBUTE_PERMISSION AttributePermission;
} EFI_BLUETOOTH_ATTRIBUTE_HEADER;
```

- **Type**: The type of this structure. It must be EFI_BLUETOOTH_UUID. See Bluetooth GATT definition. Primary Service is 0x2800. Secondary Service is 0x2801. Include Service is 0x2802. Characteristic is 0x2803. Characteristic Descriptor is 0x2900.
- **Length**: The length of this structure.
- **AttributeHandle**: The handle of the service declaration. See Bluetooth specification.
AttributePermission

The permission of the attribute. This field is only valid for the attribute of the local device. This field should be ignored for the attribute of the remote device.

//
// Bluetooth Attribute Permission
//
typedef union {
    struct {
        UINT16 Readable      : 1;
        UINT16 ReadEncryption   : 1;
        UINT16 ReadAuthentication : 1;
        UINT16 ReadAuthorization  : 1;
        UINT16 ReadKeySize     : 5;
        UINT16 Reserved1      : 7;
        UINT16 Writeable      : 1;
        UINT16 WriteEncryption   : 1;
        UINT16 WriteAuthentication : 1;
        UINT16 WriteAuthorization : 1;
        UINT16 WriteKeySize    : 5;
        UINT16 Reserved2      : 7;
    } Permission;
    UINT32 Data32;
} EFI_BLUETOOTH_ATTRIBUTE_PERMISSION;

Readable    The attribute is readable.
ReadEncryption      The encryption is required on read.
ReadAuthentication  The authentication is required on read.
ReadAuthorization   The authorization is required on read.
ReadKeySize       The size of key in bytes on read.
Writeable    The attribute is writeable.
WriteEncryption   The encryption is required on write.
WriteAuthentication  The authentication is required on write.
WriteAuthorization   The authorization is required on write.
WriteKeySize       The size of key in bytes on write.

typedef struct {
    EFI_BLUETOOTH_ATTRIBUTE_HEADER Header;
    UINT16 EndGroupHandle;
    EFI_BLUETOOTH_UUID ServiceUuid;
} EFI_BLUETOOTH_GATT_PRIMARY_SERVICE_INFO;

EndGroupHandle    The handle of the last attribute within the service definition. See Bluetooth specification.
Header            The header of this structure.
typedef struct {
  EFI_BLUETOOTH_ATTRIBUTE_HEADER Header;
  UINT16 StartGroupHandle;
  UINT16 EndGroupHandle;
  EFI_BLUETOOTH_UUID ServiceUuid;
} EFI_BLUETOOTH_GATT_INCLUDE_SERVICE_INFO;

Header
  The header of this structure.

typedef struct {
  EFI_BLUETOOTH_ATTRIBUTE_HEADER Header;
  UINT8 CharacteristicProperties;
  UINT16 CharacteristicValueHandle;
  EFI_BLUETOOTH_UUID CharacteristicUuid;
} EFI_BLUETOOTH_GATT_CHARACTERISTIC_INFO;

Header
  The header of this structure.

typedef struct {
  EFI_BLUETOOTH_ATTRIBUTE_HEADER Header;
  EFI_BLUETOOTH_UUID CharacteristicDescriptorUuid;
} EFI_BLUETOOTH_GATT_CHARACTERISTIC_DESCRIPTOR_INFO;

Header
  The header of this structure.

Status Codes Returned

<table>
<thead>
<tr>
<th>Code</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>EFI_SUCCESS</td>
<td>The Bluetooth discovered service information is returned successfully.</td>
</tr>
<tr>
<td>EFI_DEVICE_ERROR</td>
<td>A hardware error occurred trying to retrieve the Bluetooth discovered service information.</td>
</tr>
</tbody>
</table>
BLUETOOTH_ATTRIBUTE_PROTOCOL::GetDeviceInfo

Summary
Get Bluetooth device information.

Prototype
```c
typedef EFI_STATUS (EFIAPI *EFI_BLUETOOTH_ATTRIBUTE_GET_DEVICE_INFO)(
    IN EFI_BLUETOOTH_ATTRIBUTE_PROTOCOL *This,
    OUT UINTN *DeviceInfoSize,
    OUT VOID **DeviceInfo
);
```

Parameters

- **This**: Pointer to the `EFI_BLUETOOTH_ATTRIBUTE_PROTOCOL` instance.
- **DeviceInfoSize**: A pointer to the size, in bytes, of the `DeviceInfo` buffer.
- **DeviceInfo**: A pointer to a callee allocated buffer that returns Bluetooth device information. Calllee allocates this buffer by using EFI Boot Service `AllocatePool()`. If this device is Bluetooth classic device, `EFI_BLUETOOTH_DEVICE_INFO` should be used. If this device is Bluetooth LE device, `EFI_BLUETOOTH_LE_DEVICE_INFO` should be used.

Description
The `GetDeviceInfo()` function returns Bluetooth device information. The size of `DeviceInfo` structure should never be assumed and the value of `DeviceInfoSize` is the only valid way to know the size of `DeviceInfo`.

Related Definitions
```c
typedef struct {
    UINT8           Address[6];
    UINT8           Type;
} BLUETOOTH_LE_ADDRESS;

typedef struct {
    UINT32          Version;
    BLUETOOTH_LE_ADDRESS   BD_ADDR;
    BLUETOOTH_LE_ADDRESS   DirectAddress;
    UINT8           RSSI;
    UINTN           AdvertismentDataSize;
    VOID           *AdvertismentData;
} EFI_BLUETOOTH_LE_DEVICE_INFO;
```

- **Version**: The version of the structure. A value of zero represents the `EFI_BLUETOOTH_LE_DEVICE_INFO` structure as defined here. Future
version of this specification may extend this data structure in a backward compatible way and increase the value of Version.

**BD_ADDR**
48bit Bluetooth device address and 1byte address type.

**DirectAddress**
48bit random device address and 1byte address type.

**RSSI**
Bluetooth RSSI. See Bluetooth specification for detail.

**AdvertisementDataSize**
The size of AdvertisementData in bytes.

**AdvertisementData**
Bluetooth LE advertisement data. See Bluetooth specification for detail.

### Status Codes Returned

<table>
<thead>
<tr>
<th>Code</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>EFI_SUCCESS</td>
<td>The Bluetooth device information is returned successfully.</td>
</tr>
<tr>
<td>EFI_DEVICE_ERROR</td>
<td>A hardware error occurred trying to retrieve the Bluetooth device information.</td>
</tr>
</tbody>
</table>

### EFI_BLUETOOTH_ATTRIBUTE_SERVICE_BINDING_PROTOCOL

#### Summary
The EFI Bluetooth ATTRIBUTE Service Binding Protocol is used to locate EFI Bluetooth ATTRIBUTE Protocol drivers to create and destroy child of the driver to communicate with other Bluetooth device by using Bluetooth ATTRIBUTE protocol.

#### GUID
```c
#define EFI_BLUETOOTH_ATTRIBUTE_SERVICE_BINDING_PROTOCOL_GUID { \
    0x5639867a, 0x8c8e, 0x408d, 0xac, 0x2f, 0x61, 0xbd, 0xc0, 0xbb, 0xbb \
}
```

#### Description
The Bluetooth ATTRIBUTE consumer need locate EFI_BLUETOOTH_ATTRIBUTE_SERVICE_BINDING_PROTOCOL and call CreateChild() to create a new child of EFI_BLUETOOTH_ATTRIBUTE_PROTOCOL instance. Then use EFI_BLUETOOTH_ATTRIBUTE_PROTOCOL for Bluetooth communication. After use, the Bluetooth ATTRIBUTE consumer need call DestroyChild() to destroy it.

### 26.5 EFI Bluetooth LE Configuration Protocol

#### EFI_BLUETOOTH_LE_CONFIG_PROTOCOL

#### Summary
This protocol abstracts user interface configuration for BluetoothLe device.

#### GUID
#define EFI_BLUETOOTH_LE_CONFIG_PROTOCOL_GUID \
{ 0x8f76da58, 0x1f99, 0x4275, { 0xa4, 0xec, 0x47, 0x56, 0x51, 0x5b, 0x1c, 0xe8 }}

Protocol Interface Structure

typedef struct _EFI_BLUETOOTH_LE_CONFIG_PROTOCOL {
  EFI_BLUETOOTH_LE_CONFIG_INIT        Init;
  EFI_BLUETOOTH_LE_CONFIG_SCAN        Scan;
  EFI_BLUETOOTH_LE_CONFIG_CONNECT     Connect;
  EFI_BLUETOOTH_LE_CONFIG_DISCONNECT Disconnect;
  EFI_BLUETOOTH_LE_CONFIG_GET_DATA    GetData;
  EFI_BLUETOOTH_LE_CONFIG_SET_DATA    SetData;
  EFI_BLUETOOTH_LE_CONFIG_GET_REMOTE_DATA GetRemoteData;
  EFI_BLUETOOTH_LE_CONFIG_GET_REMOTE_DATA GetRemoteData;
  EFI_BLUETOOTH_LE_CONFIG_SEND_SMP_AUTH_DATA SendSmpAuthData;
  EFI_BLUETOOTH_LE_CONFIG_REGISTER_SMP_AUTH_CALLBACK RegisterSmpAuthCallback;
  EFI_BLUETOOTH_LE_CONFIG_REGISTER_SMP_SEND_SMP_AUTH_DATA SendSmpAuthData;
  EFI_BLUETOOTH_LE_CONFIG_REGISTER_SMP_SEND_SMP_DATA_CALLBACK SendSmpDataCallback;
  EFI_BLUETOOTH_LE_CONFIG_REGISTER_SMP_SET_DATA_CALLBACK SendSmpSetDataCallback;
  EFI_BLUETOOTH_LE_CONFIG_REGISTER_SMP_AUTH_CALLBACK RegisterSmpAuthCallback;
  EFI_BLUETOOTH_LE_CONFIG_REGISTER_SMP_SEND_SMP_AUTH_DATA SendSmpAuthData;
  EFI_BLUETOOTH_LE_CONFIG_REGISTER_SMP_SEND_SMP_DATA_CALLBACK SendSmpDataCallback;
  EFI_BLUETOOTH_LE_CONFIG_REGISTER_SMP_SET_DATA_CALLBACK SendSmpSetDataCallback;
  EFI_BLUETOOTH_LE_CONFIG_REGISTER_SMP_AUTH_CALLBACK RegisterSmpAuthCallback;
  EFI_BLUETOOTH_LE_CONFIG_REGISTER_SMP_SEND_SMP_AUTH_DATA SendSmpAuthData;
  EFI_BLUETOOTH_LE_CONFIG_REGISTER_SMP_SEND_SMP_DATA_CALLBACK SendSmpDataCallback;
  EFI_BLUETOOTH_LE_CONFIG_REGISTER_SMP_SET_DATA_CALLBACK SendSmpSetDataCallback;
  EFI_BLUETOOTH_LE_CONFIG_REGISTER_SMP_AUTH_CALLBACK RegisterSmpAuthCallback;
  EFI_BLUETOOTH_LE_CONFIG_REGISTER_SMP_SEND_SMP_AUTH_DATA SendSmpAuthData;
  EFI_BLUETOOTH_LE_CONFIG_REGISTER_SMP_SEND_SMP_DATA_CALLBACK SendSmpDataCallback;
  EFI_BLUETOOTH_LE_CONFIG_REGISTER_SMP_SET_DATA_CALLBACK SendSmpSetDataCallback;
} EFI_BLUETOOTH_LE_CONFIG_PROTOCOL;

Parameters

Init

  Initialize BluetoothLE host controller and local device. See the Init() function description.

Scan

  Scan BluetoothLE device. See the Scan() function description.

Connect

  Connect one BluetoothLE device. See the Connect() function description.

Disconnect

  Disconnect one BluetoothLE device. See the Disconnect() function description.

GetData

  Get BluetoothLE configuration data. See the GetData() function description.

SetData

  Set BluetoothLE configuration data. See the SetData() function description.

GetRemoteData

  Get remote BluetoothLE device data. See the GetRemoteData() function description.

RegisterSmpAuthCallback

  Register Security Manager Callback function. This function will be called from Bluetooth BUS driver whenever user interaction is required for security protocol authorization/authentication. See the RegisterSmpAuthCallback() function description.

SendSmpAuthData

  Send user input (Authentication/Authorization) such as passkey, confirmation (yes/no) in response to pairing request. See the SendSmpAuthData() function description.
RegisterSmpGetDataCallback
Register a callback function to get SMP related data. See the RegisterSmpGetDataCallback() function description.

RegisterSmpSetDataCallback
Register a callback function to set SMP related data. See the RegisterSmpGetDataCallback() function description.

RegisterLinkConnectCompleteCallback
Register link connect complete callback function. See the RegisterLinkConnectCompleteCallback() function description.

Description
The EFI_BLUETOOTH_LE_CONFIG_PROTOCOL abstracts the BluetoothLE configuration. User can use BluetoothLE configuration to interactive with BluetoothLE bus driver.

BLUETOOTH_LE_CONFIG_PROTOCOL.Init
Summary
Initialize BluetoothLE host controller and local device.

Prototype
typedef
EFI_STATUS
(EIFIAPI *EFI_BLUETOOTH_LE_CONFIG_INIT)(
    IN EFI_BLUETOOTH_LE_CONFIG_PROTOCOL *This
);

Parameters
This Pointer to the EFI_BLUETOOTH_LE_CONFIG_PROTOCOL instance.

Description
The Init() function initializes BluetoothLE host controller and local device.

Status Codes Returned
<table>
<thead>
<tr>
<th>Code</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>EFI_SUCCESS</td>
<td>The BluetoothLE host controller and local device is initialized successfully.</td>
</tr>
<tr>
<td>EFI_DEVICE_ERROR</td>
<td>A hardware error occurred trying to initialize the BluetoothLE host controller and local device.</td>
</tr>
</tbody>
</table>

BLUETOOTH_LE_CONFIG_PROTOCOL.Scan
Summary
Scan BluetoothLE device.
Prototype

typedef
EFI_STATUS
(EFI_API *EFI_BLUETOOTH_LE_CONFIG_SCAN)(
    IN EFI_BLUETOOTH_LE_CONFIG_PROTOCOL *This,
    IN BOOLEAN  ReScan,
    IN UINT32   Timeout;
    IN EFI_BLUETOOTH_LE_CONFIG_SCAN_PARAMETER *ScanParameter, OPTIONAL
    IN EFI_BLUETOOTH_LE_CONFIG_SCAN_CALLBACK_FUNCTION Callback,
    IN VOID *Context
);

Parameters

This       Pointer to the EFI_BLUETOOTH_LE_CONFIG_PROTOCOL instance.
ReScan     If TRUE, a new scan request is submitted no matter there is scan result before. If FALSE and there is scan result, the previous scan result is returned and no scan request is submitted.
Timeout    Duration in milliseconds for which to scan.
ScanParameter     If it is not NULL, the ScanParameter is used to perform a scan by the BluetoothLE bus driver. If it is NULL, the default parameter is used.
Callback    The callback function. This function is called if a BluetoothLE device is found during scan process.
Context     Data passed into Callback function. This is optional parameter and may be NULL.

Description

The Scan() function scans BluetoothLE device. When this function is returned, it just means scan request is submitted. It does not mean scan process is started or finished. Whenever there is a BluetoothLE device is found, the Callback function will be called. Callback function might be called before this function returns or after this function returns.

Related Definitions

typedef struct {
    // Scan parameter
    UINT32 Version;
    UINT8  ScanType;
    UINT16 ScanInterval;
    UINT16 ScanWindow;
    UINT8  ScanningFilterPolicy;
    // Scan result filter
    UINT8 AdvertisementFlagFilter;
} EFI_BLUETOOTH_LE_CONFIG_SCAN_PARAMETER;
Version

The version of the structure. A value of zero represents the EFI_BLUETOOTH_LE_CONFIG_SCAN_PARAMETER structure as defined here. Future version of this specification may extend this data structure in a backward compatible way and increase the value of Version.

ScanType

Passive scanning or active scanning. See Bluetooth specification.

ScanInterval

Recommended scan interval to be used while performing scan.

ScanWindow

Recommended scan window to be used while performing a scan.

ScanningFilterPolicy

Recommended scanning filter policy to be used while performing a scan.

AdvertisementFlagFilter

This is one byte flag to serve as a filter to remove unneeded scan result. For example, set BIT0 means scan in LE Limited Discoverable Mode. Set BIT1 means scan in LE General Discoverable Mode. See Supplement to Bluetooth Core Specification.

typedef

EFI_STATUS

(EIFIAPI *EFI_BLUETOOTH_LE_CONFIG_SCAN_CALLBACK_FUNCTION) (IN EFI_BLUETOOTH_LE_CONFIG_PROTOCOL *This,
IN VOID *Context,
IN EFI_BLUETOOTH_LE_SCAN_CALLBACK_INFORMATION *CallbackInfo);

This

Pointer to the EFI_BLUETOOTH_LE_CONFIG_PROTOCOL instance.

Context

Context passed from scan request.

CallbackInfo

Data related to scan result. NULL CallbackInfo means scan complete.

typedef struct{
BLUETOOTH_LE_ADDRESS BDA;address;
BLUETOOTH_LE_ADDRESS DirectAddress;
UINT8 RemoteDeviceState;
INT8 RSSI;
UINTN AdvertisementDataSize;
VOID *AdvertisementData;
} EFI_BLUETOOTH_LE_SCAN_CALLBACK_INFORMATION;

Status Codes Returned

<table>
<thead>
<tr>
<th>Status Code</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>EFI_SUCCESS</td>
<td>The Bluetooth scan request is submitted.</td>
</tr>
<tr>
<td>EFI_DEVICE_ERROR</td>
<td>A hardware error occurred trying to scan the Bluetooth device.</td>
</tr>
</tbody>
</table>

**BLUETOOTH_LE_CONFIG_PROTOCOL.Connect**

**Summary**

Connect a BluetoothLE device.
Prototype

typedef
EFI_STATUS
(EIFIAPI *EFI_BLUETOOTH_LE_CONFIG_CONNECT)(
IN EFI_BLUETOOTH_LE_CONFIG_PROTOCOL *This,
IN BOOLEAN AutoReconnect,
IN BOOLEAN DoBonding;
IN EFI_BLUETOOTH_LE_CONFIG_CONNECT_PARAMETER *ConnectParameter,
OPTIONAL
IN BLUETOOTH_LE_ADDRESS *BD_ADDR
);

Parameters

This Pointer to the EFI_BLUETOOTH_LE_CONFIG_PROTOCOL instance.
AutoReconnect If TRUE, the BluetoothLE host controller needs to do an auto reconnect. If FALSE, the BluetoothLE host controller does not do an auto reconnect.
DoBonding If TRUE, the BluetoothLE host controller needs to do a bonding. If FALSE, the BluetoothLE host controller does not do a bonding.
ConnectParameter If it is not NULL, the ConnectParameter is used to perform a scan by the BluetoothLE bus driver. If it is NULL, the default parameter is used.
BD_ADDR The address of the BluetoothLE device to be connected.

Description

The Connect() function connects a Bluetooth device. When this function is returned successfully, a new EFI_BLUETOOTH_IO_PROTOCOL is created.

Related Definitions

typedef struct {
    UINT32 Version;
    UINT16 ScanInterval;
    UINT16 ScanWindow;
    UINT16 ConnIntervalMin;
    UINT16 ConnIntervalMax;
    UINT16 ConnLatency;
    UINT16 SupervisionTimeout;
    } EFI_BLUETOOTH_LE_CONFIG_CONNECT_PARAMETER;

Version The version of the structure. A value of zero represents the EFI_BLUETOOTH_LE_CONFIG_CONNECT_PARAMETER structure as defined here. Future version of this specification may extend this data structure in a backward compatible way and increase the value of Version.

ScanInterval Recommended scan interval to be used while performing scan before connect.
**ScanWindow**
Recommended scan window to be used while performing a connection.

**ConnIntervalMin**
Minimum allowed connection interval. Shall be less than or equal to **ConnIntervalMax**.

**ConnIntervalMax**
Maximum allowed connection interval. Shall be greater than or equal to **ConnIntervalMin**.

**ConnLatency**
Slave latency for the connection in number of connection events.

**SupervisionTimeout**
Link supervision timeout for the connection.

### Status Codes Returned

<table>
<thead>
<tr>
<th>Code</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>EFI_SUCCESS</strong></td>
<td>The BluetoothLE device is connected successfully.</td>
</tr>
<tr>
<td><strong>EFI_ALREADY_STARTED</strong></td>
<td>The BluetoothLE device is already connected.</td>
</tr>
<tr>
<td><strong>EFI_NOT_FOUND</strong></td>
<td>The BluetoothLE device is not found.</td>
</tr>
<tr>
<td><strong>EFI_DEVICE_ERROR</strong></td>
<td>A hardware error occurred trying to connect the BluetoothLE device.</td>
</tr>
</tbody>
</table>

**BLUETOOTH_LE_CONFIG_PROTOCOL.Disconnect**

**Summary**
Disconnect a BluetoothLE device.

**Prototype**

```c
typedef
    EFI_STATUS
    (EFIAPI *EFI_BLUETOOTH_LE_CONFIG_DISCONNECT)(
    IN EFI_BLUETOOTH_LE_CONFIG_PROTOCOL *This,
    IN BLUETOOTH_LE_ADDRESS       *BD_ADDR,
    IN UINT8               Reason
    );
```

**Parameters**

- **This**
  Pointer to the **EFI_BLUETOOTH_LE_CONFIG_PROTOCOL** instance.
- **BD_ADDR**
  The address of BluetoothLE device to be connected.
- **Reason**
  BluetoothLE disconnect reason. See Bluetooth specification for detail.

**Description**

The **Disconnect()** function disconnects a BluetoothLE device. When this function is returned successfully, the **EFI_BLUETOOTH_ATTRIBUTE_PROTOCOL** associated with this device is destroyed and all services associated are stopped.
Status Codes Returned

<table>
<thead>
<tr>
<th>Code</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>EFI_SUCCESS</td>
<td>The BluetoothLE device is disconnected successfully.</td>
</tr>
<tr>
<td>EFI_NOT_STARTED</td>
<td>The BluetoothLE device is not connected.</td>
</tr>
<tr>
<td>EFI_NOT_FOUND</td>
<td>The BluetoothLE device is not found.</td>
</tr>
<tr>
<td>EFI_DEVICE_ERROR</td>
<td>A hardware error occurred trying to disconnect the BluetoothLE device.</td>
</tr>
</tbody>
</table>

BLUETOOTH_LE_CONFIG_PROTOCOL.GetData

Summary
Get BluetoothLE configuration data.

Prototype
```
typedef
EFI_STATUS
(EFIAPI *EFI_BLUETOOTH_LE_CONFIG_GET_DATA) (\n  IN EFI_BLUETOOTH_LE_CONFIG_PROTOCOL    *This,\n  IN EFI_BLUETOOTH_CONFIG_DATA_TYPE     DataType,\n  IN OUT UINTN               *DataSize,\n  IN OUT VOID                *Data\n);```

Parameters
- **This**: Pointer to the EFI_BLUETOOTH_LE_CONFIG_PROTOCOL instance.
- **DataType**: Configuration data type.
- **DataSize**: On input, indicates the size, in bytes, of the data buffer specified by Data. On output, indicates the amount of data actually returned.
- **Data**: A pointer to the buffer of data that will be returned.

Description
The GetData() function returns BluetoothLE configuration data. For remote BluetoothLE device configuration data, please use GetRemoteData() function with valid BD_ADDR.

Status Codes Returned

<table>
<thead>
<tr>
<th>Code</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>EFI_SUCCESS</td>
<td>The BluetoothLE configuration data is returned successfully.</td>
</tr>
<tr>
<td>EFI_INVALID_PARAMETER</td>
<td>One or more of the following conditions is TRUE:</td>
</tr>
<tr>
<td></td>
<td>- DataSize is NULL.</td>
</tr>
<tr>
<td></td>
<td>- *DataSize is 0.</td>
</tr>
<tr>
<td></td>
<td>- Data is NULL.</td>
</tr>
<tr>
<td>EFI_UNSUPPORTED</td>
<td>The DataType is unsupported.</td>
</tr>
<tr>
<td>EFI_NOT_FOUND</td>
<td>The DataType is not found.</td>
</tr>
<tr>
<td>EFI_BUFFER_TOO_SMALL</td>
<td>The buffer is too small to hold the buffer.</td>
</tr>
</tbody>
</table>
**BLUETOOTH_LE_CONFIG_PROTOCOL.SetData**

**Summary**
Set BluetoothLE configuration data.

**Prototype**
```c
typedef
EFI_STATUS
(EIFIAPI *EFI_BLUETOOTH_LE_CONFIG_SET_DATA) (  
    IN EFI_BLUETOOTH_LE_CONFIG_PROTOCOL *This,
    IN EFI_BLUETOOTH_CONFIG_DATA_TYPE DataType,
    IN UINTN DataSize,
    IN VOID *Data
);
```

**Parameters**
- **This**  
  Pointer to the `EFI_BLUETOOTH_LE_CONFIG_PROTOCOL` instance.
- **DataType**  
  Configuration data type.
- **DataSize**  
  Indicates the size, in bytes, of the data buffer specified by `Data`.
- **Data**  
  A pointer to the buffer of data that will be set.

**Description**
The `SetData()` function sets local BluetoothLE device configuration data. Not all `DataType` can be set.

**Status Codes Returned**

<table>
<thead>
<tr>
<th>Status Code</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>EFI_SUCCESS</td>
<td>The BluetoothLE configuration data is set successfully.</td>
</tr>
<tr>
<td>EFI_INVALID_PARAMETER</td>
<td>One or more of the following conditions is TRUE:</td>
</tr>
<tr>
<td></td>
<td>● <code>DataSize</code> is 0.</td>
</tr>
<tr>
<td></td>
<td>● <code>Data</code> is NULL.</td>
</tr>
<tr>
<td>EFI_UNSUPPORTED</td>
<td>The <code>DataType</code> is unsupported.</td>
</tr>
<tr>
<td>EFI_WRITE_PROTECTED</td>
<td>Cannot set configuration data.</td>
</tr>
</tbody>
</table>

**BLUETOOTH_LE_CONFIG_PROTOCOL.GetRemoteData**

**Summary**
Get remote BluetoothLE device configuration data.
Prototype

```c
typedef
  EFI_STATUS
  (EFI_API *EFI_BLUETOOTH_LE_CONFIG_GET_REMOTE_DATA) (  
    IN EFI_BLUETOOTH_LE_CONFIG_PROTOCOL    *This,
    IN EFI_BLUETOOTH_CONFIG_DATA_TYPE     DataType,
    IN BLUETOOTH_LE_ADDRESS          *BDAddr,
    IN OUT UINTN               *DataSize,
    IN OUT VOID                *Data
  );
```

Parameters

- **This**: Pointer to the `EFI_BLUETOOTH_LE_CONFIG_PROTOCOL` instance.
- **DataType**: Configuration data type.
- **BDAddr**: Remote BluetoothLE device address.
- **DataSize**: On input, indicates the size, in bytes, of the data buffer specified by `Data`. On output, indicates the amount of data actually returned.
- **Data**: A pointer to the buffer of data that will be returned.

Description

The `GetRemoteData()` function returns remote BluetoothLE device configuration data.

Status Codes Returned

<table>
<thead>
<tr>
<th>Status Code</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>EFI_SUCCESS</td>
<td>The remote BluetoothLE device configuration data is returned successfully.</td>
</tr>
<tr>
<td>EFI_INVALID_PARAMETER</td>
<td>One or more of the following conditions is TRUE:</td>
</tr>
<tr>
<td></td>
<td>- <code>DataSize</code> is NULL.</td>
</tr>
<tr>
<td></td>
<td>- <code>*DataSize</code> is 0.</td>
</tr>
<tr>
<td></td>
<td>- <code>Data</code> is NULL.</td>
</tr>
<tr>
<td>EFI_UNSUPPORTED</td>
<td>The <code>DataType</code> is unsupported.</td>
</tr>
<tr>
<td>EFI_NOT_FOUND</td>
<td>The <code>DataType</code> is not found.</td>
</tr>
<tr>
<td>EFI_BUFFER_TOO_SMALL</td>
<td>The buffer is too small to hold the buffer.</td>
</tr>
</tbody>
</table>

**BLUETOOTH_LE_CONFIG_PROTOCOL.RegisterSmpAuthCallback**

Summary

Register Security Manager Protocol callback function for user authentication/authorization.
Prototype

typedef

EFI_STATUS

(EIFIAPI *EFI_BLUETOOTH_LE_REGISTER_SMP_AUTH_CALLBACK)(
    IN EFI_BLUETOOTH_LE_CONFIG_PROTOCOL *This,
    IN EFI_BLUETOOTH_LE_SMP_CALLBACK   Callback,
    IN VOID               *Context
);

Parameters

This  Pointer to the EFI_BLUETOOTH_LE_CONFIG_PROTOCOL instance.
Callback  Callback function for user authentication/authorization.
Context  Data passed into callback function. This is optional parameter and may be NULL.

Description

The RegisterSmpAuthCallback() function register Security Manager Protocol callback function for user authentication/authorization.

Related Definitions

typedef

EFI_STATUS

(EIFIAPI *EFI_BLUETOOTH_LE_SMP_CALLBACK)  (
    IN EFI_BLUETOOTH_LE_CONFIG_PROTOCOL    *This,
    IN VOID                  *Context,
    IN BLUETOOTH_LE_ADDRESS          *BDAddr,
    IN EFI_BLUETOOTH_LE_SMP_EVENT_DATA_TYPE  EventDataType,
    IN UINTN                 DataSize,
    IN VOID                  *Data
);

This  Pointer to the EFI_BLUETOOTH_LE_CONFIG_PROTOCOL instance.
Context  Data passed into callback function. This is optional parameter and may be NULL.
BDAddr  Remote BluetoothLE device address.
EventDataType  Event data type in EFI_BLUEOTH_LE_SMP_EVENT_DATA_TYPE.
DataSize  Indicates the size, in bytes, of the data buffer specified by Data.
Data  A pointer to the buffer of data.
typedef enum {
    EfiBlutoothSmpAuthorizationRequestEvent,
    EfiBlutoothSmpPasskeyReadyEvent,
    EfiBlutoothSmpPasskeyRequestEvent,
    EfiBlutoothSmpOOBDataRequestEvent,
    EfiBlutoothSmpNumericComparisonEvent,
} EFI_BLUETOOTH_LE_SMP_EVENT_DATA_TYPE;

EfiBlutoothSmpAuthorizationRequestEvent
It indicates an authorization request. No data is associated with the callback input. In the output data, the application should return the authorization value. The data structure is BOOLEAN. TRUE means YES. FALSE means NO.

EfiBlutoothSmpPasskeyReadyEvent
It indicates that a passkey has been generated locally by the driver, and the same passkey should be entered at the remote device. The callback input data is the passkey of type UINT32, to be displayed by the application. No output data should be returned.

EfiBlutoothSmpPasskeyRequestEvent
It indicates that the driver is requesting for the passkey has been generated at the remote device. No data is associated with the callback input. The output data is the passkey of type UINT32, to be entered by the user.

EfiBlutoothSmpOOBDataRequestEvent
It indicates that the driver is requesting for the passkey that has been pre-shared out-of-band with the remote device. No data is associated with the callback input. The output data is the stored OOB data of type UINT8[16].

EfiBlutoothSmpNumericComparisonEvent
In indicates that a number have been generated locally by the bus driver, and also at the remote device, and the bus driver wants to know if the two numbers match. The callback input data is the number of type UINT32. The output data is confirmation value of type BOOLEAN. TRUE means comparison pass. FALSE means comparison fail.

Status Codes Returned

<table>
<thead>
<tr>
<th>Code</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>EFI_SUCCESS</td>
<td>The SMP callback function is registered successfully.</td>
</tr>
<tr>
<td>EFI_ALREADY_STARTED</td>
<td>A callback function is already registered on the same attribute opcode and attribute handle, when the Callback is not NULL.</td>
</tr>
<tr>
<td>EFI_NOT_STARTED</td>
<td>A callback function is not registered on the same attribute opcode and attribute handle, when the Callback is NULL.</td>
</tr>
</tbody>
</table>

BLUETOOTH_LE_CONFIG_PROTOCOL.SendSmpAuthData

Summary
Send user authentication/authorization to remote device.
Prototype

```c
typedef
  EFI_STATUS
  (EFIAPI *EFI_BLUETOOTH_LE_SEND_SMP_AUTH_DATA)(
    IN EFI_BLUETOOTH_LE_CONFIG_PROTOCOL    *This,
    IN BLUETOOTH_LE_ADDRESS          *BDAddr,
    IN EFI_BLUETOOTH_LE_SMP_EVENT_DATA_TYPE  EventDataType,
    IN UINTN                 DataSize,
    IN VOID                  *Data
  );
```

Parameters

- **This**: Pointer to the `EFI_BLUETOOTH_LE_CONFIG_PROTOCOL` instance.
- **BDAddr**: Remote BluetoothLE device address.
- **EventDataType**: Event data type in `EFI_BLUETOOTH_LE_SMP_EVENT_DATA_TYPE`.
- **DataSize**: The size of `Data` in bytes, of the data buffer specified by `Data`.
- **Data**: A pointer to the buffer of data that will be sent. The data format depends on the type of SMP event data being responded to. See `EFI_BLUETOOTH_LE_SMP_EVENT_DATA_TYPE`.

Description

The `SendSmpAuthData()` function sends user authentication/authorization to remote device. It should be used to send these information after the caller gets the request data from the callback function by `RegisterSmpAuthCallback()`.

Status Codes Returned

<table>
<thead>
<tr>
<th>Code</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>EFI_SUCCESS</td>
<td>The SMP authorization data is sent successfully.</td>
</tr>
<tr>
<td>EFI_NOT_READY</td>
<td>SMP is not in the correct state to receive the auth data</td>
</tr>
</tbody>
</table>

**BLUETOOTH_LE_CONFIG_PROTOCOL.RegisterSmpGetDataCallback**

Summary

Register a callback function to get SMP related data.
Prototype

typedef
EFI_STATUS
(EFI_API * EFI_BLUETOOTH_LE_CONFIG_REGISTER_SMP_GET_DATA_CALLBACK)(
    IN EFI_BLUETOOTH_LE_CONFIG_PROTOCOL *This,
    IN EFI_BLUETOOTH_LE_CONFIG_SMP_GET_DATA_CALLBACK Callback,
    IN VOID *Context
);

Parameters

This Pointer to the EFI_BLUETOOTH_LE_CONFIG_PROTOCOL instance.
Callback Callback function for SMP get data.
Context Data passed into callback function. This is optional parameter and may be NULL.

Description

The RegisterSmpGetDataCallback() function registers a callback function to get SMP related data.

Related Definitions

typedef
EFI_STATUS
(EFI_API * EFI_BLUETOOTH_LE_CONFIG_SMP_GET_DATA_CALLBACK)(
    IN EFI_BLUETOOTH_LE_CONFIG_PROTOCOL *This,
    IN VOID *Context,
    IN BLUETOOTH_LE_ADDRESS *BDAddr,
    IN EFI_BLUETOOTH_LE_SMP_DATA_TYPE DataType,
    IN OUT UINTN *DataSize,
    OUT VOID *Data
);

This Pointer to the EFI_BLUETOOTH_LE_CONFIG_PROTOCOL instance.
Context Data passed into callback function. This is optional parameter and may be NULL.
BDAddr Remote BluetoothLE device address. For Local device setting, it should be NULL.
DataType Data type in EFI_BLUETOOTH_LE_SMP_DATA_TYPE.
DataSize On input, indicates the size, in bytes, of the data buffer specified by Data. On output, indicates the amount of data actually returned.
Data A pointer to the buffer of data that will be returned.

typedef enum {
    // For local device only
    EfiBluetoothSmpLocalIR, /* If Key hierarchy is supported */
    EfiBluetoothSmpLocalER, /* If Key hierarchy is supported */
EfiBlutoothSmpLocalDHK, /* If Key hierarchy is supported. OPTIONAL */

// For peer specific
EfiBlutoothSmpKeysDistributed = 0x1000,
EfiBlutoothSmpKeySize,
EfiBlutoothSmpKeyType,
EfiBlutoothSmpPeerLTK,
EfiBlutoothSmpPeerIRK,
EfiBlutoothSmpPeerCSRK,
EfiBlutoothSmpPeerRand,
EfiBlutoothSmpPeerEDIV,
EfiBlutoothSmpPeerSignCounter,
EfiBlutoothSmpLocalLTK, /* If Key hierarchy not supported */
EfiBlutoothSmpLocalIRK, /* If Key hierarchy not supported */
EfiBlutoothSmpLocalCSRK, /* If Key hierarchy not supported */
EfiBlutoothSmpLocalSignCounter,
EfiBlutoothSmpLocalDIV,
EfiBlutoothSmpPeerAddressList,
EfiBlutoothSmpMax,
} EFI_BLUETOOTH_LE_SMP_DATA_TYPE;

EfiBlutoothSmpLocalIR
It is a 128-bit Identity Root (IR) key to generate IRK. Data structure is UINT8[16]. See Bluetooth specification. This is only required when Bluetooth key hierarchy is supported. This type is for the local device only.

EfiBlutoothSmpLocalER
It is a 128-bit Encryption Root (ER) key to generate LTK and CSRK. Data structure is UINT8[16]. See Bluetooth specification. This is only required when Bluetooth key hierarchy is supported. This type is for the local device only.

EfiBlutoothSmpLocalDHK
It is a 128-bit Diversifier Hiding Key (DHK) to generate EDIV. Data structure is UINT8[16]. See Bluetooth specification. This is only required when Bluetooth key hierarchy is supported. This type is for the local device only.

EfiBlutoothSmpKeysDistributed
It is LE Key Distribution Format. Data structure is UINT8. See Bluetooth specification. This is the peer device specific information.

EfiBlutoothSmpKeySize
It indicates the size of keys in bytes. It is the negotiated key size between local device and peer device. Data structure is UINTN. This is the peer device specific information.
EfiBlutoothSmpKeyType

Indicates support for MITM/Secure connection. It is the negotiated Authentication Requirements between local device and peer device. See Bluetooth Spec 3.H.3.5.1. Data structure is UINT8. BIT0: MITM, BIT1: SC. This is the peer device specific information.

EfiBlutoothSmpPeerLTK

It is a 128-bit Long-Term Key (LTK) to generate the contributory session key for an encrypted connection. Data structure is UINT8[16]. See Bluetooth specification. This is the peer device specific information.

EfiBlutoothSmpPeerIRK

It is a 128-bit Identity Resolving Key (IRK) to generate and resolve random addresses. Data structure is UINT8[16]. See Bluetooth specification. This is the peer device specific information.

EfiBlutoothSmpPeerCSRK

It is a 128-bit Connection-Signature Resolving Key (CSRK) to sign data and verify signatures on the receiving device. Data structure is UINT8[16]. See Bluetooth specification. This is the peer device specific information.

EfiBlutoothSmpPeerRand

It is a 64-bit Random number (Rand) to identify the LTK distributed during LE legacy pairing. Data structure is UINT64. See Bluetooth specification. This is the peer device specific information.

EfiBlutoothSmpPeerEDIV

It is a 16-bit Encrypted Diversifier (EDIV) to identify the LTK distributed during LE legacy pairing. Data structure is UINT16. See Bluetooth specification. This is the peer device specific information.

EfiBlutoothSmpPeerSignCounter

It is a 32-bit Sign Counter to assist MAC generation. Data structure is UINT32. See Bluetooth specification. This is the peer device specific information.

EfiBlutoothSmpLocalLTK

It is a 128-bit Long-Term Key (LTK) to generate the contributory session key for an encrypted connection. Data structure is UINT8[16]. See Bluetooth specification. This is only required when Bluetooth key hierarchy is not supported. This is the peer specific local device information.

EfiBlutoothSmpLocalIRK

It is a 128-bit Identity Resolving Key (IRK) to generate and resolve random addresses. Data structure is UINT8[16]. See Bluetooth specification. This is only required when Bluetooth key hierarchy is not supported. This is the peer specific local device information.
EfiBlutoothSmpLocalCSRK

It is a 128-bit Connection-Signature Resolving Key (CSRK) to sign data and verify signatures on the receiving device. Data structure is `UINT8[16]`. See Bluetooth specification. This is only required when Bluetooth key hierarchy is not supported. This is the peer specific local device information.

EfiBlutoothSmpLocalSignCounter

It is a 32-bit Sign Counter to assist MAC generation. Data structure is `UINT32`. See Bluetooth specification. This is the peer specific local device information.

EfiBlutoothSmpLocalDIV

It is a 16-bit Diversifier (DIV) to be used as index to recover LTK. Data structure is `UINT16`. See Bluetooth specification. This is the peer specific local device information.

EfiBlutoothSmpPeerAddressList

A list of Bluetooth peer addresses that have been connected before. The data structure is `BLUETOOTH_LE_ADDRESS[]`. The data size must be a multiple of `sizeof(BLUETOOTH_LE_ADDRESS)`.

Status Codes Returned

<table>
<thead>
<tr>
<th>Code</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>EFI_SUCCESS</td>
<td>The SMP get data callback function is registered successfully.</td>
</tr>
<tr>
<td>EFI_ALREADY_STARTED</td>
<td>A callback function is already registered on the same attribute opcode and attribute handle, when the Callback is not NULL.</td>
</tr>
<tr>
<td>EFI_NOT_STARTED</td>
<td>A callback function is not registered on the same attribute opcode and attribute handle, when the Callback is NULL.</td>
</tr>
</tbody>
</table>

BLUETOOTH_LE_CONFIG_PROTOCOL.RegisterSmpSetDataCallback

Summary

Register a callback function to set SMP related data.

Prototype

```c
typedef
EFI_STATUS
(EFIAPI * EFI_BLUETOOTH_LE_CONFIG_REGISTER_SMP_SET_DATA_CALLBACK
)(
  IN EFI_BLUETOOTH_LE_CONFIG_PROTOCOL       *This,
  IN EFI_BLUETOOTH_LE_CONFIG_SMP_SET_DATA_CALLBACK Callback,
  IN VOID                     *Context
);
```

Parameters

- **This**: Pointer to the `EFI_BLUETOOTH_LE_CONFIG_PROTOCOL` instance.
- **Callback**: Callback function for SMP set data.
Context
Data passed into callback function. This is optional parameter and may be NULL.

Description
The `RegisterSmpSetDataCallback()` function registers a callback function to set SMP related data.

Related Definitions

```
typedef
EFI_STATUS
(EIFIAPI * EFI_BLUETOOTH_LE_CONFIG_SMP_SET_DATA_CALLBACK)  (
    IN EFI_BLUETOOTH_LE_CONFIG_PROTOCOL *This,
    IN VOID               *Context,
    IN BLUETOOTH_LE_ADDRESS       *BDAddr,
    IN EFI_BLUETOOTH_LE_SMP_DATA_TYPE  Type,
    IN UINTN               DataSize,
    IN VOID               *Data
);
```

This
Pointer to the `EFI_BLUETOOTH_LE_CONFIG_PROTOCOL` instance.

Context
Data passed into callback function. This is optional parameter and may be NULL.

BDAddr
Remote BluetoothLE device address.

DataType
Data type in `EFI_BLUETOOTH_LE_SMP_DATA_TYPE`.

DataSize
Indicates the size, in bytes, of the data buffer specified by `Data`.

Data
A pointer to the buffer of data.

Status Codes Returned

<table>
<thead>
<tr>
<th>Status Code</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>EFI_SUCCESS</td>
<td>The SMP get data callback function is registered successfully.</td>
</tr>
<tr>
<td>EFI_ALREADY_STARTED</td>
<td>A callback function is already registered on the same attribute opcode and attribute handle, when the Callback is not NULL.</td>
</tr>
<tr>
<td>EFI_NOT_STARTED</td>
<td>A callback function is not registered on the same attribute opcode and attribute handle, when the Callback is NULL.</td>
</tr>
</tbody>
</table>

**BLUETOOTH_LE_CONFIG_PROTOCOL/RegisterLinkConnectCompleteCallback**

Summary
Register link connect complete callback function.
Prototype

typedef
EFI_STATUS
(EFIAPI *EFI_BLUETOOTH_LE_CONFIG_REGISTER_CONNECT_COMPLETE_CALLBACK) (  
    IN EFI_BLUETOOTH_LE_CONFIG_PROTOCOL                   *This,
    IN EFI_BLUETOOTH_LE_CONFIG_CONNECT_COMPLETE_CALLBACK   Callback,
    IN VOID                                                *Context
    );

Parameters

This Pointer to the EFI_BLUETOOTH_LE_CONFIG_PROTOCOL instance.
Callback The callback function. NULL means unregister.
Context Data passed into Callback function. This is optional parameter and may be NULL.

Description

The RegisterLinkConnectCompleteCallback() function registers Bluetooth link connect complete callback function. The Bluetooth Configuration driver may call RegisterLinkConnectCompleteCallback() to register a callback function. During pairing, Bluetooth bus driver must trigger this callback function to report device state, if it is registered. Then Bluetooth Configuration driver will get information on device connection, according to CallbackType defined by EFI_BLUETOOTH_CONNECT_COMPLETE_CALLBACK_TYPE.

Related Definitions

typedef
EFI_STATUS
(EFIAPI *EFI_BLUETOOTH_LE_CONFIG_CONNECT_COMPLETE_CALLBACK) (  
    IN EFI_BLUETOOTH_LE_CONFIG_PROTOCOL                   *This,
    IN VOID                                          *Context,
    IN EFI_BLUETOOTH_CONNECT_COMPLETE_CALLBACK_TYPE  CallbackType,
    IN BLUETOOTH_LE_ADDRESS                          *BDAddr,
    IN VOID                                          *InputBuffer,
    IN UINTN                                         InputBufferSize
    );

This Pointer to the EFI_BLUETOOTH_LE_CONFIG_PROTOCOL instance.
Context Context passed from registration.
CallbackType Callback type in EFI_BLUETOOTH_CONNECT_COMPLETE_CALLBACK_TYPE.
BDAddr A pointer to BluetoothLE device address.
InputBuffer A pointer to the buffer of data that is input from callback caller.
InputBufferSize Indicates the size, in bytes, of the data buffer specified by InputBuffer.
### Status Codes Returned

<table>
<thead>
<tr>
<th>Code</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>EFI_SUCCESS</td>
<td>The link connect complete callback function is registered successfully.</td>
</tr>
<tr>
<td>EFI_ALREADY_STARTED</td>
<td>A callback function is already registered on the same attribute opcode and attribute handle, when the Callback is not NULL.</td>
</tr>
<tr>
<td>EFI_NOT_STARTED</td>
<td>A callback function is not registered on the same attribute opcode and attribute handle, when the Callback is NULL.</td>
</tr>
</tbody>
</table>
27 - Network Protocols — VLAN, EAP, Wi-Fi and Supplicant

27.1 VLAN Configuration Protocol

EFI_VLAN_CONFIG_PROTOCOL

Summary
This protocol is to provide manageability interface for VLAN configuration.

GUID
#define EFI_VLAN_CONFIG_PROTOCOL_GUID
{0x9e23d768, 0xd2f3, 0x4366,
{0x9f, 0xc3, 0x3a, 0x7a, 0xba, 0x86, 0x43, 0x74}}

Protocol Interface Structure
typedef struct _EFI_VLAN_CONFIG_PROTOCOL {
  EFI_VLAN_CONFIG_SET         Set;
  EFI_VLAN_CONFIG_FIND        Find;
  EFI_VLAN_CONFIG_REMOVE      Remove;
} EFI_VLAN_CONFIG_PROTOCOL;

Parameters
- **Set**: Create new VLAN device or modify configuration parameter of an already-configured VLAN.
- **Find**: Find configuration information for specified VLAN or all configured VLANs.
- **Remove**: Remove a VLAN device.

Description
This protocol is to provide manageability interface for VLAN setting. The intended VLAN tagging implementation is IEEE802.1Q.

EFI_VLAN_CONFIG_PROTOCOL.Set ()

Summary
Create a VLAN device or modify the configuration parameter of an already-configured VLAN.
Prototype

typedef
EFI_STATUS
(EIFIAPI * EFI_VLAN_CONFIG_SET) (  
    IN EFI_VLAN_CONFIG_PROTOCOL      *This,
    IN UINT16                      VlanId,
    IN UINT8                       Priority
);

Parameters

This Pointer to EFI_VLAN_CONFIG_PROTOCOL instance.
VlanId A unique identifier (1-4094) of the VLAN which is being created or modified, or zero (0).
Priority 3 bit priority in VLAN header. Priority 0 is default value. If VlanId is zero (0), Priority is ignored.

Description

The Set() function is used to create a new VLAN device or change the VLAN configuration parameters. If the VlanId hasn’t been configured in the physical Ethernet device, a new VLAN device will be created. If a VLAN with this VlanId is already configured, then related configuration will be updated as the input parameters.

If VlanId is zero, the VLAN device will send and receive untagged frames. Otherwise, the VLAN device will send and receive VLAN-tagged frames containing the VlanId.

If VlanId is out of scope of (0-4094), EFI_INVALID_PARAMETER is returned.

If Priority is out of the scope of (0-7), then EFI_INVALID_PARAMETER is returned.

If there is not enough system memory to perform the registration, then EFI_OUT_OF_RESOURCES is returned.

Status Codes Returned

<table>
<thead>
<tr>
<th>Status Code</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>EFI_SUCCESS</td>
<td>The VLAN is successfully configured</td>
</tr>
<tr>
<td>EFI_INVALID_PARAMETER</td>
<td>One or more of following conditions is TRUE</td>
</tr>
<tr>
<td></td>
<td>• This is NULL</td>
</tr>
<tr>
<td></td>
<td>• VlanId is an invalid VLAN Identifier</td>
</tr>
<tr>
<td></td>
<td>• Priority is invalid</td>
</tr>
<tr>
<td>EFI_OUT_OF_RESOURCES</td>
<td>There is not enough system memory to perform the registration.</td>
</tr>
</tbody>
</table>

EFI_VLAN_CONFIG_PROTOCOL.Find()

Summary

Find configuration information for specified VLAN or all configured VLANs.
Prototype

```c
typedef EFI_STATUS
  (EFIAPI *EFI_VLAN_CONFIG_FIND) (
    IN EFI_VLAN_CONFIG_PROTOCOL  *This,
    IN UINT16           *VlanId, OPTIONAL
    OUT UINT16           *NumberOfVlan,
    OUT EFI_VLAN_FIND_DATA     **Entries
  );
```

Parameters

- **This**: Pointer to `EFI_VLAN_CONFIG_PROTOCOL` instance.
- **VlanId**: Pointer to VLAN identifier. Set to `NULL` to find all configured VLANs
- **NumberOfVlan**: The number of VLANs which is found by the specified criteria
- **Entries**: The buffer which receive the VLAN configuration. Type `EFI_VLAN_FIND_DATA` is defined below.

Description

The `Find()` function is used to find the configuration information for matching VLAN and allocate a buffer into which those entries are copied.

Related Definitions

```c
//EFI_VLAN_FIND_DATA

typedef struct {
  UINT16 VlanId;
  UINT8  Priority;
} EFI_VLAN_FIND_DATA;
```

- **VlanId**: Vlan Identifier
- **Priority**: Priority of this VLAN

Status Codes Returned

<table>
<thead>
<tr>
<th>Code</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>EFI_SUCCESS</td>
<td>The VLAN is successfully found</td>
</tr>
<tr>
<td>EFI_INVALID_PARAMETER</td>
<td>One or more of following conditions is <strong>TRUE</strong></td>
</tr>
<tr>
<td></td>
<td>• <strong>This</strong> is <strong>NULL</strong></td>
</tr>
<tr>
<td></td>
<td>• Specified <strong>VlanId</strong> is invalid</td>
</tr>
<tr>
<td>EFI_NOT_FOUND</td>
<td>No matching VLAN is found</td>
</tr>
</tbody>
</table>
**EFI_VLAN_CONFIG_PROTOCOL.Remove ()**

**Summary**
Remove the configured VLAN device

**Prototype**

```c
typedef EFI_STATUS (EFIAPI * EFI_VLAN_CONFIG_REMOVE) (  
    IN EFI_VLAN_CONFIG_PROTOCOL *This,  
    IN UINT16 VlanId
);
```

**Parameters**

- **This**: Pointer to `EFI_VLAN_CONFIG_PROTOCOL` instance.
- **VlanId**: Identifier (0-4094) of the VLAN to be removed.

**Description**

The `Remove()` function is used to remove the specified VLAN device. If the `VlanId` is out of the scope of (0-4094), `EFI_INVALID_PARAMETER` is returned. If specified VLAN hasn’t been previously configured, `EFI_NOT_FOUND` is returned.

**Status Codes Returned**

<table>
<thead>
<tr>
<th>Status Code</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>EFI_SUCCESS</code></td>
<td>The VLAN is successfully removed</td>
</tr>
<tr>
<td><code>EFI_INVALID_PARAMETER</code></td>
<td>One or more of following conditions is <code>TRUE</code></td>
</tr>
<tr>
<td></td>
<td>• <code>This</code> is <code>NULL</code></td>
</tr>
<tr>
<td></td>
<td>• <code>VlanId</code> is an invalid parameter.</td>
</tr>
<tr>
<td><code>EFI_NOT_FOUND</code></td>
<td>The to-be-removed VLAN does not exist</td>
</tr>
</tbody>
</table>

### 27.2 EAP Protocol

This section defines the EAP protocol. This protocol is designed to make the EAP framework configurable and extensible. It is intended for the supplicant side.

**EFI_EAP_PROTOCOL**

**Summary**

This protocol is used to abstract the ability to configure and extend the EAP framework.
GUID
#define EFI_EAP_PROTOCOL_GUID
{ 0x5d9f96db, 0xe731, 0x4caa,
{0xa0, 0x0d, 0x72, 0xe1, 0x87, 0xcd, 0x77, 0x62 } }

Protocol Interface Structure
typedef struct _EFI_EAP_PROTOCOL {
 EFI_EAP_SET_DESIRED_AUTHENTICATION_METHOD SetDesiredAuthMethod;
 EFI_EAP_REGISTER_AUTHENTICATION_METHOD RegisterAuthMethod;
} EFI_EAP_PROTOCOL;

Parameters
 SetDesiredAuthMethod Set the desired EAP authentication method for the Port. See the
 SetDesiredAuthMethod() function description.
 RegisterAuthMethod Register an EAP authentication method. See the
 RegisterAuthMethod() function description.

Description
EFI_EAP_PROTOCOL is used to configure the desired EAP authentication method for the EAP framework
and extend the EAP framework by registering new EAP authentication method on a Port. The EAP
framework is built on a per-Port basis. Herein, a Port means a NIC. For the details of EAP protocol, please
refer to RFC 2284.

Related Definitions
//
// Type for the identification number assigned to the Port by the // System in
which the Port resides.
//
typedef VOID * EFI_PORT_HANDLE;

EFI_EAP.SetDesiredAuthMethod()

Summary
Set the desired EAP authentication method for the Port.

Prototype
typedef
EFI_STATUS
(EIFIAP *EFI_EAP_SET_DESIRED_AUTHENTICATION_METHOD) (  
IN struct _EFI_EAP_PROTOCOL *This,
IN UINT8 EapAuthType
);

Parameters
This A pointer to the EFI_EAP_PROTOCOL instance that indicates the
calling context. Type EFI_EAP_PROTOCOL is defined in Section 1.1.
EapAuthType

The type of the desired EAP authentication method for the Port. It should be the type value defined by RFC. See RFC 2284 for details. Current valid values are defined in “Related Definitions”.

Related Definitions

```c
//
// EAP Authentication Method Type (RFC 3748)
//
#define EFI_EAP_TYPE_TLS 13 /* REQUIRED - RFC 5216 */
```

Description

The `SetDesiredAuthMethod()` function sets the desired EAP authentication method indicated by `EapAuthType` for the Port.

If `EapAuthType` is an invalid EAP authentication type, then `EFI_INVALID_PARAMETER` is returned.

If the EAP authentication method of `EapAuthType` is unsupported, then it will return `EFI_UNSUPPORTED`.

The cryptographic strength of `EFI_EAP_TYPE_TLS` shall be at least of hash strength SHA-256 and RSA key length of at least 2048 bits.

Status Codes Returned

<table>
<thead>
<tr>
<th>Code</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>EFI_SUCCESS</td>
<td>The desired EAP authentication method is set successfully.</td>
</tr>
<tr>
<td>EFI_INVALID_PARAMETER</td>
<td><code>EapAuthType</code> is an invalid EAP authentication type.</td>
</tr>
<tr>
<td>EFI_UNSUPPORTED</td>
<td>The EAP authentication method of <code>EapAuthType</code> is unsupported by the Port.</td>
</tr>
</tbody>
</table>

**EFI_EAP.RegisterAuthMethod()**

Summary

Register an EAP authentication method.

Prototype

```c
typedef
EFI_STATUS
(EFIAPI *EFI_EAP_REGISTER_AUTHENTICATION_METHOD) (  
    IN struct _EFI_EAP_PROTOCOL   *This,
    IN UINT8                      EapAuthType,
    IN EFI_EAP_BUILD_RESPONSE_PACKET Handler
    );
```

Parameters

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>This</td>
<td>A pointer to the <code>EFI_EAP_PROTOCOL</code> instance that indicates the calling context. Type <code>EFI_EAP_PROTOCOL</code> is defined in Section 1.1.</td>
</tr>
<tr>
<td>EapAuthType</td>
<td>The type of the EAP authentication method to register. It should be the type value defined by RFC. See RFC 2284 for details. Current valid</td>
</tr>
</tbody>
</table>
values are defined in the `SetDesiredAuthMethod()` function
description.

**Handler**
The handler of the EAP authentication method to register. Type
`EFI_EAP_BUILD_RESPONSE_PACKET` is defined in “Related
Definitions”.

**Related Definitions**

typedef

`EFI_STATUS`

`(EFI_API *EFI_EAP_BUILD_RESPONSE_PACKET) (``

`IN EFI_PORT_HANDLE` PortNumber

`IN UINT8` *RequestBuffer,

`IN UINTN` RequestSize,

`IN UINT8` *Buffer,

`IN OUT UINTN` *BufferSize

```

/*++

Routine Description:
Build EAP response packet in response to the EAP request packet specified by
(RequestBuffer, RequestSize).

Arguments:
PortNumber   - Specified the Port where the EAP request packet comes.
RequestBuffer  - Pointer to the most recently received EAP-Request packet.
RequestSize   - Packet size in bytes for the most recently received EAP-
Request packet.
Buffer     - Pointer to the buffer to hold the built packet.
BufferSize   - Pointer to the buffer size in bytes. On input, it is the
buffer size provided by the caller. On output, it is the buffer size in fact
needed to contain the packet.

Returns:
EFI_SUCCESS   - The required EAP response packet is built successfully.
others     - Failures are encountered during the packet building process.

--*/
```

**Description**
The `RegisterAuthMethod()` function registers the user provided EAP authentication method, the type
of which is `EapAuthType` and the handler of which is `Handler`.

If `EapAuthType` is an invalid EAP authentication type, then `EFI_INVALID_PARAMETER` is returned.

If there is not enough system memory to perform the registration, then `EFI_OUT_OF_RESOURCES` is
returned.
## Status Codes Returned

<table>
<thead>
<tr>
<th>Code</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>EFI_SUCCESS</td>
<td>The EAP authentication method of <code>EapAuthType</code> is registered successfully.</td>
</tr>
<tr>
<td>EFI_INVALID_PARAMETER</td>
<td><code>EapAuthType</code> is an invalid EAP authentication type.</td>
</tr>
<tr>
<td>EFI_OUT_OF_RESOURCES</td>
<td>There is not enough system memory to perform the registration.</td>
</tr>
</tbody>
</table>

### 27.2.1 EAPManagement Protocol

This section defines the EAP management protocol. This protocol is designed to provide ease of management and ease of test for EAPOL state machine. It is intended for the supplicant side. It conforms to IEEE 802.1x specification.

**EFI_EAP_MANAGEMENT_PROTOCOL**

**Summary**

This protocol provides the ability to configure and control EAPOL state machine, and retrieve the status and the statistics information of EAPOL state machine.

**GUID**

```
#define EFI_EAP_MANAGEMENT_PROTOCOL_GUID {
  0xbb62e663, 0x625d, 0x40b2,
  { 0xa0, 0x88, 0xbb, 0xe8, 0x36, 0x23, 0xa2, 0x45 }
```

**Protocol Interface Structure**

```
typedef struct _EFI_EAP_MANAGEMENT_PROTOCOL {
  EFI_EAP_GET_SYSTEM_CONFIGURATION  GetSystemConfiguration;
  EFI_EAP_SET_SYSTEM_CONFIGURATION  SetSystemConfiguration;
  EFI_EAP_INITIALIZE_PORT          InitializePort;
  EFI_EAP_USER_LOGON               UserLogon;
  EFI_EAP_USER_LOGOFF              UserLogoff;
  EFI_EAP_GET_SUPPLICANT_STATUS    GetSupplicantStatus;
  EFI_EAP_SET_SUPPLICANT_CONFIGURATION  SetSupplicantConfiguration;
  EFI_EAP_GET_SUPPLICANT_STATISTICS GetSupplicantStatistics;
} EFI_EAP_MANAGEMENT_PROTOCOL;
```

**Parameters**

- **GetSystemConfiguration** Read the system configuration information associated with the Port. See the `GetSystemConfiguration()` function description.
- **SetSystemConfiguration** Set the system configuration information associated with the Port. See the `SetSystemConfiguration()` function description.
- **InitializePort** Cause the EAPOL state machines for the Port to be initialized. See the `InitializePort()` function description.
- **UserLogon** Notify the EAPOL state machines for the Port that the user of the System has logged on. See the `UserLogon()` function description.
UserLogoff

Notify the EAPOL state machines for the Port that the user of the System has logged off. See the UserLogoff() function description.

GetSupplicantStatus

Read the status of the Supplicant PAE state machine for the Port, including the current state and the configuration of the operational parameters. See the GetSupplicantStatus() function description.

SetSupplicantConfiguration

Set the configuration of the operational parameter of the Supplicant PAE state machine for the Port. See the SetSupplicantConfiguration() function description.

GetSupplicantStatistics

Read the statistical information regarding the operation of the Supplicant associated with the Port. See the GetSupplicantStatistics() function description.

Description

The EFI_EAP_MANAGEMENT protocol is used to control, configure and monitor EAPOL state machine on a Port. EAPOL state machine is built on a per-Port basis. Herein, a Port means a NIC. For the details of EAPOL, please refer to IEEE 802.1x specification.

EFI_EAP_MANAGEMENT.GetSystemConfiguration()

Summary

Read the system configuration information associated with the Port.

Prototype

typedef

EFI_STATUS

(EIFIAPI *EFI_EAP_GET_SYSTEM_CONFIGURATION) (  
IN struct _EFI_EAP_MANAGEMENT_PROTOCOL *This,  
OUT BOOLEAN               *SystemAuthControl,  
OUT EFI_EAPOL_PORT_INFO         *PortInfo OPTIONAL  
);

Parameters

This  

A pointer to the EFI_EAP_MANAGEMENT_PROTOCOL instance that indicates the calling context. Type EFI_EAP_MANAGEMENT_PROTOCOL is defined in <Hyperlink>Section 27.2.1.

SystemAuthControl  

Returns the value of the SystemAuthControl parameter of the System. TRUE means Enabled. FALSE means Disabled.

PortInfo  

Returns EFI_EAPOL_PORT_INFO structure to describe the Port's information. This parameter can be NULL to ignore reading the Port's information. Type EFI_EAPOL_PORT_INFO is defined in “Related Definitions”.


Related Definitions

//
// PAE Capabilities
//
#define PAE_SUPPORT_AUTHENTICATOR 0x01
#define PAE_SUPPORT_SUPPLICANT 0x02

typedef struct _EFI_EAPOL_PORT_INFO {
  EFI_PORT_HANDLE     PortNumber;
  UINT8                ProtocolVersion;
  UINT8                PaeCapabilities;
} EFI_EAPOL_PORT_INFO;

PortNumber The identification number assigned to the Port by the System in which the Port resides.
ProtocolVersion The protocol version number of the EAPOL implementation supported by the Port.
PaeCapabilities The capabilities of the PAE associated with the Port. This field indicates whether Authenticator functionality, Supplicant functionality, both, or neither, is supported by the Port’s PAE.

Description

The GetSystemConfiguration() function reads the system configuration information associated with the Port, including the value of the SystemAuthControl parameter of the System is returned in SystemAuthControl and the Port’s information is returned in the buffer pointed to by PortInfo. The Port’s information is optional. If PortInfo is NULL, then reading the Port’s information is ignored.

If SystemAuthControl is NULL, then EFI_INVALID_PARAMETER is returned.

Status Codes Returned

<table>
<thead>
<tr>
<th>Code</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>EFI_SUCCESS</td>
<td>The system configuration information of the Port is read successfully.</td>
</tr>
<tr>
<td>EFI_INVALID_PARAMETER</td>
<td>SystemAuthControl is NULL.</td>
</tr>
</tbody>
</table>

EFI_EAP_MANAGEMENT.SetSystemConfiguration()

Summary

Set the system configuration information associated with the Port.

Prototype

typedef

EFI_STATUS

(EIFIAPI *EFI_EAP_SET_SYSTEM_CONFIGURATION) ( 
IN struct _EFI_EAP_MANAGEMENT_PROTOCOL *This, 

IN BOOLEAN SystemAuthControl
);

Parameters

This A pointer to the EFI_EAP_MANAGEMENT_PROTOCOL instance that indicates the calling context. Type EFI_EAP_MANAGEMENT_PROTOCOL is defined in <Hyperlink>Section 27.2.1.

SystemAuthControl The desired value of the SystemAuthControl parameter of the System. TRUE means Enabled. FALSE means Disabled.

Description

The SetSystemConfiguration() function sets the value of the SystemAuthControl parameter of the System to SystemAuthControl.

Status Codes Returned

| EFI_SUCCESS | The system configuration information of the Port is set successfully. |

---

EFI_EAP_MANAGEMENT.InitializePort()

Summary

Cause the EAPOL state machines for the Port to be initialized.

Prototype

typedef EFI_STATUS (EFIAPI *EFI_EAP_INITIALIZE_PORT) (
    IN struct _EFI_EAP_MANAGEMENT_PROTOCOL *This
);

Parameters

This A pointer to the EFI_EAP_MANAGEMENT_PROTOCOL instance that indicates the calling context. Type EFI_EAP_MANAGEMENT_PROTOCOL is defined in <Hyperlink>Section 27.2.1.

Description

The InitializePort() function causes the EAPOL state machines for the Port.

Status Codes Returned

| EFI_SUCCESS | The Port is initialized successfully. |
EFI_EAP_MANAGEMENT.UserLogon()

Summary
Notify the EAPOL state machines for the Port that the user of the System has logged on.

Prototype
typedef
EFI_STATUS
(EIFIAPI *EFI_EAP_USER_LOGON) (  
    IN struct _EFI_EAP_MANAGEMENT_PROTOCOL *This,  
);

Parameters
This
A pointer to the EFI_EAP_MANAGEMENT_PROTOCOL instance that indicates the calling context. Type EFI_EAP_MANAGEMENT_PROTOCOL is defined in <Hyperlink>Section 27.2.1.

Description
The UserLogon() function notifies the EAPOL state machines for the Port.

Status Codes Returned
| EFI_SUCCESS | The Port is notified successfully. |

EFI_EAP_MANAGEMENT.UserLogoff()

Summary
Notify the EAPOL state machines for the Port that the user of the System has logged off.

Prototype
typedef
EFI_STATUS
(EIFIAPI *EFI_EAP_USER_LOGOFF) (  
    IN struct _EFI_EAP_MANAGEMENT_PROTOCOL *This,  
);

Parameters
This
A pointer to the EFI_EAP_MANAGEMENT_PROTOCOL instance that indicates the calling context. Type EFI_EAP_MANAGEMENT_PROTOCOL is defined in <Hyperlink>Section 27.2.1.

Description
The UserLogoff() function notifies the EAPOL state machines for the Port.
Status Codes Returned

<table>
<thead>
<tr>
<th>Code</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>EFI_SUCCESS</td>
<td>The Port is notified successfully.</td>
</tr>
</tbody>
</table>

EFI_EAP_MANAGEMENT.GetSupplicantStatus()

Summary

Read the status of the Supplicant PAE state machine for the Port, including the current state and the configuration of the operational parameters.

Prototype

```c
typedef EFI_STATUS (EFIAPI *EFI_EAP_GET_SUPPLICANT_STATUS) (
    IN struct _EFI_EAP_MANAGEMENT_PROTOCOL *This,
    OUT EFI_EAPOL_SUPPLICANT_PAE_STATE *CurrentState,
    IN OUT EFI_EAPOL_SUPPLICANT_PAE_CONFIGURATION *Configuration
);
```

Parameters

- **This**: A pointer to the `EFI_EAP_MANAGEMENT_PROTOCOL` instance that indicates the calling context. Type `EFI_EAP_MANAGEMENT_PROTOCOL` is defined in <Hyperlink>Section 27.2.1.</Hyperlink>

- **CurrentState**: Returns the current state of the Supplicant PAE state machine for the Port. Type `EFI_EAPOL_SUPPLICANT_PAE_STATE` is defined in “Related Definitions”.

- **Configuration**: Returns the configuration of the operational parameters of the Supplicant PAE state machine for the Port as required. This parameter can be NULL to ignore reading the configuration. On input, `Configuration.ValidFieldMask` specifies the operational parameters to be read. On output, `Configuration` returns the configuration of the required operational parameters. Type `EFI_EAPOL_SUPPLICANT_PAE_CONFIGURATION` is defined in “Related Definitions”.

EFI_SUCCESS The Port is notified successfully.
Related Definitions

//
// Supplicant PAE state machine (IEEE Std 802.1X Section 8.5.10)
//
typedef enum _EFI_EAPOL_SUPPLICANT_PAE_STATE {
  Logoff,
  Disconnected,
  Connecting,
  Acquired,
  Authenticating,
  Held,
  Authenticated,
  MaxSupplicantPaeState
} EFI_EAPOL_SUPPLICANT_PAE_STATE;

//
// Definitions for ValidFieldMask
//
#define AUTH_PERIOD_FIELD_VALID 0x01
#define HELD_PERIOD_FIELD_VALID 0x02
#define START_PERIOD_FIELD_VALID 0x04
#define MAX_START_FIELD_VALID 0x08

typedef struct _EFI_EAPOL_SUPPLICANT_PAE_CONFIGURATION {
  UINT8    ValidFieldMask;
  UINTN    AuthPeriod;
  UINTN    HeldPeriod;
  UINTN    StartPeriod;
  UINTN    MaxStart;
} EFI_EAPOL_SUPPLICANT_PAE_CONFIGURATION;

ValidFieldMask Indicates which of the following fields are valid.
AuthPeriod The initial value for the authWhile timer. Its default value is 30 s.
HeldPeriod The initial value for the heldWhile timer. Its default value is 60 s.
StartPeriod The initial value for the startWhen timer. Its default value is 30 s.
MaxStart The maximum number of successive EAPOL-Start messages will be sent before the Supplicant assumes that there is no Authenticator present. Its default value is 3.

Description

The GetSupplicantStatus() function reads the status of the Supplicant PAE state machine for the Port, including the current state CurrentState and the configuration of the operational parameters Configuration. The configuration of the operational parameters is optional. If Configuration is NULL, then reading the configuration is ignored. The operational parameters in Configuration to be read can also be specified by Configuration.ValidFieldMask.
If `CurrentState` is **NULL**, then **EFI_INVALID_PARAMETER** is returned.

### Status Codes Returned

<table>
<thead>
<tr>
<th>Code</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>EFI_SUCCESS</td>
<td>The status of the Supplicant PAE state machine for the Port is read successfully.</td>
</tr>
<tr>
<td>EFI_INVALID_PARAMETER</td>
<td><code>CurrentState</code> is <strong>NULL</strong>.</td>
</tr>
</tbody>
</table>

**EFI_EAP_MANAGEMENT.SetSupplicantConfiguration()**

**Summary**

Set the configuration of the operational parameter of the Supplicant PAE state machine for the Port.

**Prototype**

```c
typedef EFI_STATUS (EFIAPI *EFI_EAP_SET_SUPPLICANT_CONFIGURATION) (
    IN struct _EFI_EAP_MANAGEMENT_PROTOCOL   *This,
    IN EFI_EAPOL_SUPPLICANT_PAE_CONFIGURATION  *Configuration
);
```

**Parameters**

- **This**
  A pointer to the **EFI_EAP_MANAGEMENT_PROTOCOL** instance that indicates the calling context. Type **EFI_EAP_MANAGEMENT_PROTOCOL** is defined in `<Hyperlink>Section 27.2.1</Hyperlink>`.

- **Configuration**
  The desired configuration of the operational parameters of the Supplicant PAE state machine for the Port as required. Type **EFI_EAPOL_SUPPLICANT_PAE_CONFIGURATION** is defined in the **GetSupplicantStatus()** function description.

**Description**

The **SetSupplicantConfiguration()** function sets the configuration of the operational parameter of the Supplicant PAE state machine for the Port to **Configuration**. The operational parameters in **Configuration** to be set can be specified by **Configuration.ValidFieldMask**.

If **Configuration** is **NULL**, then **EFI_INVALID_PARAMETER** is returned.

### Status Codes Returned

<table>
<thead>
<tr>
<th>Code</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>EFI_SUCCESS</td>
<td>The configuration of the operational parameter of the Supplicant PAE state machine for the Port is set successfully.</td>
</tr>
<tr>
<td>EFI_INVALID_PARAMETER</td>
<td><strong>Configuration</strong> is <strong>NULL</strong>.</td>
</tr>
</tbody>
</table>
**EFI_EAP_MANAGEMENT.GetSupplicantStatistics()**

**Summary**

Read the statistical information regarding the operation of the Supplicant associated with the Port.

**Prototype**

```c
typedef EFI_STATUS
    (EFIAPI *EFI_EAP_GET_SUPPLICANT_STATISTICS) (
    IN struct _EFI_EAP_MANAGEMENT_PROTOCOL   *This,
    OUT EFI_EAPOL_SUPPLICANT_PAE_STATISTICS   *Statistics
    );
```

**Parameters**

- **This**  
  A pointer to the `EFI_EAP_MANAGEMENT_PROTOCOL` instance that indicates the calling context. Type `EFI_EAP_MANAGEMENT_PROTOCOL` is defined in `<Hyperlink>Section 27.2.1</Hyperlink>`.

- **Statistics**  
  Returns the statistical information regarding the operation of the Supplicant for the Port. Type `EFI_EAPOL_SUPPLICANT_PAE_STATISTICS` is defined in “Related Definitions”.

**Related Definitions**

```c
//
// Supplicant Statistics (IEEE Std 802.1X Section 9.5.2)
//
typedef struct _EFI_EAPOL_SUPPLICANT_PAE_STATISTICS {
    UINTN   EapolFramesReceived;
    UINTN   EapolFramesTransmitted;
    UINTN   EapolStartFramesTransmitted;
    UINTN   EapolLogoffFramesTransmitted;
    UINTN   EapReqIdFramesReceived;
    UINTN   EapRequestFramesReceived;
    UINTN   InvalidEapolFramesReceived;
    UINTN   EapLengthErrorFramesReceived;
    UINTN   LastEapolFrameVersion;
    UINTN   LastEapolFrameSource;
} EFI_EAPOL_SUPPLICANT_PAE_STATISTICS;
```

**EapolFramesReceived**

The number of EAPOL frames of any type that have been received by this Supplicant.
### EapolFramesTransmitted
The number of EAPOL frames of any type that have been transmitted by this Supplicant.

### EapolStartFramesTransmitted
The number of EAPOL Start frames that have been transmitted by this Supplicant.

### EapolLogoffFramesTransmitted
The number of EAPOL Logoff frames that have been transmitted by this Supplicant.

### EapRespIdFramesTransmitted
The number of EAP Resp/Id frames that have been transmitted by this Supplicant.

### EapResponseFramesTransmitted
The number of valid EAP Response frames (other than Resp/Id frames) that have been transmitted by this Supplicant.

### EapReqIdFramesReceived
The number of EAP Req/Id frames that have been received by this Supplicant.

### EapRequestFramesReceived
The number of EAP Request frames (other than Rq/Id frames) that have been received by this Supplicant.

### InvalidEapolFramesReceived
The number of EAPOL frames that have been received by this Supplicant in which the frame type is not recognized.

### EapLengthErrorFramesReceived
The number of EAPOL frames that have been received by this Supplicant in which the Packet Body Length field (7.5.5) is invalid.

### LastEapolFrameVersion
The protocol version number carried in the most recently received EAPOL frame.

### LastEapolFrameSource
The source MAC address carried in the most recently received EAPOL frame.

### Description
The `GetSupplicantStatistics()` function reads the statistical information `Statistics` regarding the operation of the Supplicant associated with the Port.

If `Statistics` is `NULL`, then `EFI_INVALID_PARAMETER` is returned.

### Status Codes Returned

<table>
<thead>
<tr>
<th>Status Code</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>EFI_SUCCESS</td>
<td>The statistical information regarding the operation of the Supplicant for the Port is read successfully.</td>
</tr>
<tr>
<td>EFI_INVALID_PARAMETER</td>
<td><code>Statistics</code> is <code>NULL</code>.</td>
</tr>
</tbody>
</table>
27.2.2 EFI EAP Management2 Protocol

**EFI_EAP_MANAGEMENT2_PROTOCOL**

**Summary**

This protocol provides the ability to configure and control EAPOL state machine, and retrieve the information, status and the statistics information of EAPOL state machine.

**GUID**

```c
#define EFI_EAP_MANAGEMENT2_PROTOCOL_GUID
   { 0x5e93c847, 0x456d, 0x40b3, 
   { 0xa6, 0xb4, 0x78, 0xb0, 0xc9, 0xcf, 0x7f, 0x20 }}
```

**Protocol Interface Structure**

```c
typedef struct _EFI_EAP_MANAGEMENT2_PROTOCOL {
    ....... // Same as EFI_EAP_MANAGEMENT_PROTOCOL
    EFI_EAP_GET_KEY GetKey;
} EFI_EAP_MANAGMENT2_PROTOCOL;
```

**Parameters**

- **GetKey**
  
  Provide Key information parsed from EAP packet. See the `GetKey()` function description.

**Description**

The **EFI_EAP_MANAGEMENT2_PROTOCOL** is used to control, configure and monitor EAPOL state machine on a Port, and return information of the Port. EAPOL state machine is built on a per-Port basis. Herein, a Port means a NIC. For the details of EAPOL, please refer to IEEE 802.1x specification.

**EFI_EAP_MANAGEMENT2_PROTOCOL.GetKey()**

**Summary**

Return key generated through EAP process.

**Prototype**

```c
typedef
EFI_STATUS
(EIFIAPI *EFI_EAP_GET_KEY)(
    IN EFI_EAP_MANAGEMENT2_PROTOCOL *This,
    IN OUT UINT8 *Msk,
    IN OUT UINTN *MskSize,
    IN OUT UINT8 *Emsk,
    IN OUT UINT8 *EmskSize
    );
```
Parameters

<table>
<thead>
<tr>
<th>Pointer to the EFI_EAP_MANAGEMENT2_PROTOCOL instance.</th>
</tr>
</thead>
<tbody>
<tr>
<td>Msk (Master Session Key) buffer.</td>
</tr>
<tr>
<td>MSK buffer size.</td>
</tr>
<tr>
<td>EMSK (Extended Master Session Key) buffer.</td>
</tr>
<tr>
<td>EMSK buffer size.</td>
</tr>
</tbody>
</table>

Description

The GetKey() function returns the key generated through EAP process, so that the 802.11 MAC layer driver can use MSK to derive more keys, e.g. PMK (Pairwise Master Key).

Status Codes Returned

<table>
<thead>
<tr>
<th>Code</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>EFI_SUCCESS</td>
<td>The operation completed successfully.</td>
</tr>
<tr>
<td>EFI_INVALID_PARAMETER</td>
<td>One or more of the following conditions is TRUE:</td>
</tr>
<tr>
<td>Msk is NULL.</td>
<td></td>
</tr>
<tr>
<td>MskSize is NULL.</td>
<td></td>
</tr>
<tr>
<td>Emsk is NULL.</td>
<td></td>
</tr>
<tr>
<td>EmskSize is NULL.</td>
<td></td>
</tr>
<tr>
<td>EFI_NOT_READY</td>
<td>MSK and EMSK are not generated in current session yet.</td>
</tr>
</tbody>
</table>

27.2.3 EFI EAP Configuration Protocol

EFI_EAP_CONFIGURATION_PROTOCOL

Summary

This protocol provides a way to set and get EAP configuration.

GUID

```
#define EFI_EAP_CONFIGURATION_PROTOCOL_GUID \
{ 0xe5b58dbb, 0x7688, 0x44b4, \ 
{ 0x97, 0xbf, 0x5f, 0x1d, 0x4b, 0xc8, 0xdb }}
```

Protocol Interface Structure

```
typedef struct _EFI_EAP_CONFIGURATION_PROTOCOL {
  EFI_EAP_CONFIGURATION_SET_DATA     SetData;
  EFI_EAP_CONFIGURATION_GET_DATA     GetData;
} EFI_EAP_CONFIGURATION_PROTOCOL;
```

Parameters

<table>
<thead>
<tr>
<th>SetData</th>
<th>Set EAP configuration data. See the SetData() function description.</th>
</tr>
</thead>
<tbody>
<tr>
<td>GetData</td>
<td>Get EAP configuration data. See the GetData() function description.</td>
</tr>
</tbody>
</table>
Description

The `EFI_EAP_CONFIGURATION_PROTOCOL` is designed to provide a way to set and get EAP configuration, such as Certificate, private key file.

### EFI_EAP_CONFIGURATION_PROTOCOL.SetData()

**Summary**

Set EAP configuration data.

**Prototype**

```c
typedef EFI_STATUS
    (EFIAPIC *EFI_EAP_CONFIGURATION_SET_DATA)(
    IN EFI_EAP_CONFIGURATION_PROTOCOL *This,
    IN EFI_EAP_TYPE EapType,
    IN EFI_EAP_CONFIG_DATA_TYPE DataType,
    IN VOID *Data,
    IN UINTN DataSize
    );
```

**Parameters**

- **This** Pointer to the `EFI_EAP_CONFIGURATION_PROTOCOL` instance.
- **EapType** EAP type. See `EFI_EAP_TYPE`.
- **DataType** Configuration data type. See `EFI_EAP_CONFIG_DATA_TYPE`.
- **Data** Pointer to configuration data.
- **DataSize** Total size of configuration data.

**Description**

The `SetData()` function sets EAP configuration to non-volatile storage or volatile storage.
Related Definitions

    //
    // Make sure it not conflict with any real EapTypeXXX
    //
    #define EFI_EAP_TYPE_ATTRIBUTE 0

    typedef enum {
        // EFI_EAP_TYPE_ATTRIBUTE
        EfiEapConfigEapAuthMethod,
        EfiEapConfigEapSupportedAuthMethod,
        // EapTypeIdentity
        EfiEapConfigIdentityString,
        // EapTypeEAPTLS/EapTypePEAP
        EfiEapConfigEapTlsCACert,
        EfiEapConfigEapTlsClientCert,
        EfiEapConfigEapTlsClientPrivateKeyFile,
        EfiEapConfigEapTlsClientPrivateKeyFilePassword,
        // ASCII format, Volatile
        EfiEapConfigEapTlsCipherSuite,
        EfiEapConfigEapTlsSupportedCipherSuite,
        // EapTypeMSChapV2
        EfiEapConfigEapMSChapV2Password, // UNICODE format, Volatile
        // EapTypePEAP
        EfiEapConfigEap2ndAuthMethod,
        // More...
    } EFI_EAP_CONFIG_DATA_TYPE;

    //
    // EFI_EAP_TYPE
    //
    typedef UINT8 EFI_EAP_TYPE;
    #define EFI_EAP_TYPE_ATTRIBUTE 0
    #define EFI_EAP_TYPE_IDENTITY 1
    #define EFI_EAP_TYPE_NOTIFICATION 2
    #define EFI_EAP_TYPE_NAK 3
    #define EFI_EAP_TYPE_MD5CHALLENGE 4
    #define EFI_EAP_TYPE_OTP 5
    #define EFI_EAP_TYPE_GTC 6
    #define EFI_EAP_TYPE_EAPTLS 13
    #define EFI_EAP_TYPE_EAPSIM 18
    #define EFI_EAP_TYPE_TTLS 21
    #define EFI_EAP_TYPE_PEAP 25
    #define EFI_EAP_TYPE_MSCHAPV2 26
    #define EFI_EAP_TYPE_EAP_EXTENSION 33
    ......
Status Codes Returned

<table>
<thead>
<tr>
<th>Status Code</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>EFI_SUCCESS</td>
<td>The EAP configuration data is set successfully.</td>
</tr>
<tr>
<td>EFI_INVALID_PARAMETER</td>
<td>One or more of the following conditions is TRUE:</td>
</tr>
<tr>
<td></td>
<td>• Data is NULL.</td>
</tr>
<tr>
<td></td>
<td>• DataSize is 0.</td>
</tr>
<tr>
<td>EFI_UNSUPPORTED</td>
<td>The EapType or DataType is unsupported.</td>
</tr>
<tr>
<td>EFI_OUT_OF_RESOURCES</td>
<td>Required system resources could not be allocated.</td>
</tr>
</tbody>
</table>

**EFI_EAP_CONFIGURATION_PROTOCOL.GetData()**

**Summary**

Get EAP configuration data.

**Prototype**

```c
typedef EFI_STATUS
(EFI_API *EFI_EAP_CONFIGURATION_PROTOCOL *This,
 IN EFI_EAP_TYPE EapType,
 IN EFI_EAP_CONFIG_DATA_TYPE DataType,
 IN OUT VOID *Data,
 IN OUT UINTN *DataSize
);
```

**Parameters**

- **This**  
  Pointer to the `EFI_EAP_CONFIGURATION_PROTOCOL` instance.
- **EapType**  
  EAP type. See `EFI_EAP_TYPE`.
- **DataType**  
  Configuration data type. See `EFI_EAP_CONFIG_DATA_TYPE`.
- **Data**  
  Pointer to configuration data.
- **DataSize**  
  Total size of configuration data. On input, it means the size of `Data` buffer. On output, it means the size of copied `Data` buffer if `EFI_SUCCESS`, and means the size of desired `Data` buffer if `EFI_BUFFER_TOO_SMALL`.

**Description**
The \texttt{GetData()} function gets EAP configuration.

- Status Codes Returned

<table>
<thead>
<tr>
<th>Code</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>EFI_SUCCESS</td>
<td>The EAP configuration data is got successfully.</td>
</tr>
<tr>
<td>EFI_INVALID_PARAMETER</td>
<td>One or more of the following conditions is TRUE:</td>
</tr>
<tr>
<td></td>
<td>- \texttt{Data} is NULL.</td>
</tr>
<tr>
<td></td>
<td>- \texttt{DataSize} is NULL.</td>
</tr>
<tr>
<td>EFI_UNSUPPORTED</td>
<td>The \texttt{EapType} or \texttt{DataType} is unsupported.</td>
</tr>
<tr>
<td>EFI_NOT_FOUND</td>
<td>The EAP configuration data is not found.</td>
</tr>
<tr>
<td>EFI_BUFFER_TOO_SMALL</td>
<td>The buffer is too small to hold the buffer.</td>
</tr>
</tbody>
</table>

27.3 EFI Wireless MAC Connection Protocol

\texttt{EFI_WIRELESS_MAC_CONNECTION_PROTOCOL}

Summary

This protocol provides management service interfaces of 802.11 MAC layer. It is used by network applications (and drivers) to establish wireless connection with an access point (AP).

GUID

\begin{verbatim}
#define EFI_WIRELESS_MAC_CONNECTION_PROTOCOL_GUID \{ 0xda55bc9, 0x45f8, 0x4bb4, \{ 0x87, 0x19, 0x52, 0x24, 0xf1, 0x8a, 0x4d, 0x45 \}
\end{verbatim}

Protocol Interface Structure

\begin{verbatim}
typedef struct _EFI_WIRELESS_MAC_CONNECTION_PROTOCOL {
  EFI_WIRELESS_MAC_CONNECTION.Scan Scan;
  EFI_WIRELESS_MAC_CONNECTION.Associate Associate;
  EFI_WIRELESS_MAC_CONNECTION.Disassociate Disassociate;
  EFI_WIRELESS_MAC_CONNECTION.Authenticate Authenticate;
  EFI_WIRELESS_MAC_CONNECTION.Deauthenticate Deauthenticate;
} EFI_WIRELESS_MAC_CONNECTION_PROTOCOL;
\end{verbatim}

Parameters

- \texttt{Scan} Determine the characteristics of the available BSSs. See the \texttt{Scan()} function description.

- \texttt{Associate} Places an association request with a specific peer MAC entity. See the \texttt{Associate()} function description.

- \texttt{Disassociate} Reports a disassociation with a specific peer MAC entity. See the \texttt{Disassociate()} function description.

- \texttt{Authenticate} Requests authentication with a specific peer MAC entity. See the \texttt{Authenticate()} function description.

- \texttt{Deauthenticate} Invalidates an authentication relationship with a peer MAC entity. See the \texttt{Deauthenticate()} function description.
Description
The **EFI_WIRELESS_MAC_CONNECTION_PROTOCOL** is designed to provide management service interfaces for the EFI wireless network stack to establish wireless connection with AP. An EFI Wireless MAC Connection Protocol instance will be installed on each communication device that the EFI wireless network stack runs on.

**EFI_WIRELESS_MAC_CONNECTION_PROTOCOL.Scan()**

Summary
Request a survey of potential BSSs that administrator can later elect to try to join.

Prototype
```
typedef
  EFI_STATUS
(EFI_API *EFI_WIRELESS_MAC_CONNECTION_SCAN)(
  IN EFI_WIRELESS_MAC_CONNECTION_PROTOCOL *This,
  IN EFI_80211_SCAN_DATA_TOKEN *Data
);
```

Parameters
- **This** Pointer to the **EFI_WIRELESS_MAC_CONNECTION_PROTOCOL** instance.
- **Data** Pointer to the scan token. Type **EFI_80211_SCAN_DATA_TOKEN** is defined in “Related Definitions” below.

Description
The **Scan()** function returns the description of the set of BSSs detected by the scan process. Passive scan operation is performed by default.

Related Definitions
```
 targetType
  //**********************************************
  //  EFI_80211_SCAN_DATA_TOKEN
  //**********************************************

typedef struct {
  EFI_EVENT  Event;
  EFI_STATUS Status;
  EFI_80211_SCAN_DATA    *Data;
  EFI_80211_SCAN_RESULT_CODE ResultCode;
  EFI_80211_SCAN_RESULT   *Result;
} EFI_80211_SCAN_DATA_TOKEN;
```

- **Event** This **Event** will be signaled after the **Status** field is updated by the EFI Wireless MAC Connection Protocol driver. The type of **Event** must be ** EFI_NOTIFY_SIGNAL**.

- **Status** Will be set to one of the following values:
**EFI_SUCCESS**: Scan operation completed successfully.

**EFI_NOT_FOUND**: Failed to find available BSS.

**EFI_DEVICE_ERROR**: An unexpected network or system error occurred.

**EFI_ACCESS_DENIED**: The scan operation is not completed due to some underlying hardware or software state.

**EFI_NOT_READY**: The scan operation is started but not yet completed.

**Data**
Pointer to the scan data. Type **EFI_80211_SCAN_DATA** is defined below.

**ResultCode**
Indicates the scan state. Type **EFI_80211_SCAN_RESULT_CODE** is defined below.

**Result**
Indicates the scan result. It is caller’s responsibility to free this buffer. Type **EFI_80211_SCAN_RESULT** is defined below.

The **EFI_80211_SCAN_DATA_TOKEN** structure is defined to support the process of determining the characteristics of the available BSSs. As input, the **Data** field must be filled in by the caller of EFI Wireless MAC Connection Protocol. After the scan operation completes, the EFI Wireless MAC Connection Protocol driver updates the **Status**, **ResultCode** and **Result** field and the **Event** is signaled.

```c
// ****************************************************************************
// EFI_80211_SCAN_DATA
// ****************************************************************************
typedef struct {
    EFI_80211_BSS_TYPE BSSType;
    EFI_80211_MAC_ADDRESS BSSID;
    UINT8 SSIDLlen;
    UINT8 *SSID;
    BOOLEAN PassiveMode;
    UINT32 ProbeDelay;
    UINT32 *ChannelList;
    UINT32 MinChannelTime;
    UINT32 MaxChannelTime;
    EFI_80211_ELEMENT_REQ *RequestInformation;
    EFI_80211_ELEMENT_SSID *SSIDList;
    EFI_80211_ACC_NET_TYPE AccessNetworkType;
    UINT8 *VendorSpecificInfo;
} EFI_80211_SCAN_DATA;
```

**BSSType**
Determines whether infrastructure BSS, IBSS, MBSS, or all, are included in the scan. Type **EFI_80211_BSS_TYPE** is defined below.

**BSSID**
Indicates a specific or wildcard BSSID. Use all binary 1s to represent all SSIDs. Type **EFI_80211_MAC_ADDRESS** is defined below.

**SSIDLen**
Length in bytes of the **SSID**. If **zero**, ignore **SSID** field.
SSId Specifies the desired SSId or the wildcard SSId. Use NULL to represent all SSIDs.

PassiveMode Indicates passive scanning if TRUE.

ProbeDelay The delay in microseconds to be used prior to transmitting a Probe frame during active scanning. If zero, the value can be overridden by an implementation-dependent default value.

ChannelList Specifies a list of channels that are examined when scanning for a BSS. If set to NULL, all valid channels will be scanned.

MinChannelTime Indicates the minimum time in TU to spend on each channel when scanning. If zero, the value can be overridden by an implementation-dependent default value.

MaxChannelTime Indicates the maximum time in TU to spend on each channel when scanning. If zero, the value can be overridden by an implementation-dependent default value.

RequestInformation Points to an optionally present element. This is an optional parameter and may be NULL. Type EFI_80211_ELEMENT_REQ is defined below.

SSIDList Indicates one or more SSID elements that are optionally present. This is an optional parameter and may be NULL. Type EFI_80211_ELEMENT_SSID is defined below.

AccessNetworkType Specifies a desired specific access network type or the wildcard access network type. Use 15 as wildcard access network type. Type EFI_80211_ACC_NET_TYPE is defined below.

VendorSpecificInfo Specifies zero or more elements. This is an optional parameter and may be NULL.

//**********************************************
// EFI_80211_BSS_TYPE
//**********************************************
typedef enum {
    IeeeInfrastructureBSS,
    IeeeIndependentBSS,
    IeeeMeshBSS,
    IeeeAnyBss
} EFI_80211_BSS_TYPE;

The EFI_80211_BSS_TYPE is defined to enumerate BSS type.

//**********************************************
// EFI_80211_MAC_ADDRESS
//**********************************************
typedef struct {
    UINT8 Addr[6];
} EFI_80211_MAC_ADDRESS;
The **EFI_80211_MAC_ADDRESS** is defined to record a 48-bit MAC address.

```c
//******************************************************************************
// EFI_80211_ELEMENT_REQ
//******************************************************************************
typedef struct {
    EFI_80211_ELEMENT_HEADER Hdr;
    UINT8 RequestIDs[1];
} EFI_80211_ELEMENT_REQ;

Hdr      Common header of an element. Type EFI_80211_ELEMENT_HEADER is defined below.
RequestIDs Start of elements that are requested to be included in the Probe Response frame. The elements are listed in order of increasing element ID.

//******************************************************************************
// EFI_80211_ELEMENT_HEADER
//******************************************************************************
typedef struct {
    UINT8 ElementID;
    UINT8 Length;
} EFI_80211_ELEMENT_HEADER;

ElementID  A unique element ID defined in IEEE 802.11 specification.
Length      Specifies the number of octets in the element body.

//******************************************************************************
// EFI_80211_ELEMENT_SSID
//******************************************************************************
typedef struct {
    EFI_80211_ELEMENT_HEADER Hdr;
    UINT8 SSId[32];
} EFI_80211_ELEMENT_SSID;

Hdr      Common header of an element.
SSId      Service set identifier. If **Hdr.Length** is zero, this field is ignored.
typedef enum {
  IeeePrivate      = 0,
  IeeePrivatewithGuest = 1,
  IeeeChargeablePublic = 2,
  IeeeFreePublic    = 3,
  IeeePersonal     = 4,
  IeeeEmergencyServOnly = 5,
  IeeeTestOrExp     = 14,
  IeeeWildcard     = 15
} EFI_80211_ACC_NET_TYPE;

The **EFI_80211_ACC_NET_TYPE** records access network types defined in IEEE 802.11 specification.

typedef enum {
  ScanSuccess,  
  ScanNotSupported
} EFI_80211_SCAN_RESULT_CODE;

ScanSuccess The scan operation finished successfully.
ScanNotSupported The scan operation is not supported in current implementation.

typedef struct {
  UINTN  NumOfBSSDesp;
  EFI_80211_BSS_DESCRIPTION **BSSDespSet;
  UINTN  NumofBSSDespFromPilot;
  EFI_80211_BSS_DESP_PILOT **BSSDespFromPilotSet;
  UINT8   *VendorSpecificInfo;
} EFI_80211_SCAN_RESULT;

NumOfBSSDesp The number of **EFI_80211_BSS_DESCRIPTION** in BSSDespSet. If zero, BSSDespSet should be ignored.

BSSDespSet Points to zero or more instances of **EFI_80211_BSS_DESCRIPTION**. Type **EFI_80211_BSS_DESCRIPTION** is defined below.

NumOfBSSDespFromPilot The number of **EFI_80211_BSS_DESP_PILOT** in BSSDespFromPilotSet. If zero, BSSDespFromPilotSet should be ignored.
BSSDespFromPilotSetPoints to zero or more instances of EFI_80211_BSS_DESP_PILOT. Type EFI_80211_BSS_DESP_PILOT is defined below.

VendorSpecificInfoSpecifies zero or more elements. This is an optional parameter and may be NULL.

// **************************************************************
// EFI_80211_BSS_DESCRIPTION
// **************************************************************
typedef struct {
    EFI_80211_MAC_ADDRESS   BSSID;
    UINT8                   *SSID;
    UINT8                   SSIDLen;
    EFI_80211_BSS_TYPE      BSSType;
    UINT16                  BeaconPeriod;
    UINT64                  Timestamp;
    UINT16                  CapabilityInfo;
    UINT8                   *BSSBasicRateSet;
    UINT8                   *OperationalRateSet;
    EFI_80211_ELEMENT_COUNTRY  *Country;
    EFI_80211_ELEMENT_RSN   RSN;
    UINT8                   RSSI;
    UINT8                   RCPIMeasurement;
    UINT8                   RSNIMeasurement;
    UINT8                   *RequestedElements;
    UINT8                   *BSSMembershipSelectorSet;
    EFI_80211_ELEMENT_EXT_CAP *ExtCapElement;
} EFI_80211_BSS_DESCRIPTION;

BSSID      Indicates a specific BSSID of the found BSS.
SSID       Specifies the SSID of the found BSS. If NULL, ignore SSIDLen field.
SSIDLen    Length in bytes of the SSID. If zero, ignore SSID field.
BSSType    Specifies the type of the found BSS.
BeaconPeriod The beacon period in TU of the found BSS.
Timestamp  The timestamp of the received frame from the found BSS.
CapabilityInfo The advertised capabilities of the BSS.
BSSBasicRateSet The set of data rates that shall be supported by all STAs that desire to join this BSS.
OperationalRateSet The set of data rates that the peer STA desires to use for communication within the BSS.
Country    The information required to identify the regulatory domain in which the peer STA is located. Type EFI_80211_ELEMENT_COUNTRY is defined below.
RSN
The cipher suites and AKM suites supported in the BSS. Type
EFI_80211_ELEMENT_RSN is defined below.

RSSI
Specifies the RSSI of the received frame.

RCPIMeasurement
Specifies the RCPI of the received frame.

RSNIMeasurement
Specifies the RSNI of the received frame.

RequestedElements
Specifies the elements requested by the request element of the
Probe Request frame. This is an optional parameter and may be
NULL.

BSSMembershipSelectorSet
Specifies the BSS membership selectors that represent the
set of features that shall be supported by all STAs to join this BSS.

ExtCapElement
Specifies the parameters within the Extended Capabilities element
that are supported by the MAC entity. This is an optional parameter
and may be NULL. Type EFI_80211_ELEMENT_EXT_CAP is defined
below.

//******************************************************************************
// EFI_80211_ELEMENT_COUNTRY
//******************************************************************************

typedef struct {
    EFI_80211_ELEMENT_HEADER Hdr;
    UINT8         CountryStr[3];
    EFI_80211_COUNTRY_TRIPLET CountryTriplet[1];
} EFI_80211_ELEMENT_COUNTRY;

Hdr         Common header of an element.
CountryStr   Specifies country strings in 3 octets.
CountryTriplet Indicates a triplet that repeated in country element. The number of
triplets is determined by the Hdr.Length field.

//******************************************************************************
// EFI_80211_COUNTRY_TRIPLET
//******************************************************************************

typedef union {
    EFI_80211_COUNTRY_TRIPLET_SUBBAND Subband;
    EFI_80211_COUNTRY_TRIPLET_OPERATE Operating;
} EFI_80211_COUNTRY_TRIPLET;

Subband       The subband triplet.
Operating      The operating triplet.
/**********************************************
// EFI_80211_COUNTRY_TRIPLET_SUBBAND
//**********************************************
typedef struct {
  UINT8 FirstChannelNum;
  UINT8 NumOfChannels;
  UINT8 MaxTxPowerLevel;
} EFI_80211_COUNTRY_TRIPLET_SUBBAND;

FirstChannelNum      Indicates the lowest channel number in the subband. It has a positive integer value less than 201.
NumOfChannels        Indicates the number of channels in the subband.
MaxTxPowerLevel      Indicates the maximum power in dBm allowed to be transmitted.

//**********************************************
// EFI_80211_COUNTRY_TRIPLET_OPERATE
//**********************************************
typedef struct {
  UINT8 OperatingExtId;
  UINT8 OperatingClass;
  UINT8 CoverageClass;
} EFI_80211_COUNTRY_TRIPLET_OPERATE;

OperatingExtId       Indicates the operating extension identifier. It has a positive integer value of 201 or greater.
OperatingClass       Index into a set of values for radio equipment set of rules.
CoverageClass        Specifies aAirPropagationTime characteristics used in BSS operation. Refer the definition of aAirPropagationTime in IEEE 802.11 specification.

//**********************************************
// EFI_80211_ELEMENT_RSN
//**********************************************
typedef struct {
  EFI_80211_ELEMENT_HEADER    Hdr;
  EFI_80211_ELEMENT_DATA_RSN  *Data;
} EFI_80211_ELEMENT_RSN;

Hdr                   Common header of an element.
Data                  Points to RSN element. Type EFI_80211_ELEMENT_DATA_RSN is defined below. The size of a RSN element is limited to 255 octets.
EFI_80211_ELEMENT_DATA_RSN

typedef struct {
    UINT16    Version;
    UINT32    GroupDataCipherSuite;
    //UINT16   PairwiseCipherSuiteCount;
    //UINT32   PairwiseCipherSuiteList[PairwiseCipherSuiteCount];
    //UINT16   AKMSuiteCount;
    //UINT32   AKMSuiteList[AKMSuiteCount];
    //UINT16   RSNCapabilities;
    //UINT16   PMKIDCount;
    //UINT8    PMKIDList[PMKIDCount][16];
    //UINT32   GroupManagementCipherSuite;
} EFI_80211_ELEMENT_DATA_RSN;

Volume indicates the version number of the RSNA protocol. Value 1 is defined in current IEEE 802.11 specification.

GroupDataCipherSuite specifies the cipher suite selector used by the BSS to protect group address frames.

PairwiseCipherSuiteCount indicates the number of pairwise cipher suite selectors that are contained in PairwiseCipherSuiteList.

PairwiseCipherSuiteList contains a series of cipher suite selectors that indicate the pairwise cipher suites contained in this element.

AKMSuiteCount indicates the number of AKM suite selectors that are contained in AKMSuiteList.

AKMSuiteList contains a series of AKM suite selectors that indicate the AKM suites contained in this element.

RSNCapabilities indicates requested or advertised capabilities.

PMKIDCount indicates the number of PKMIDs in the PMKIDList.

PMKIDList contains zero or more PKMIDs that the STA believes to be valid for the destination AP.

GroupManagementCipherSuite specifies the cipher suite selector used by the BSS to protect group addressed robust management frames.

EFI_80211_ELEMENT_EXT_CAP

typedef struct {
    EFI_80211_ELEMENT_HEADER Hdr;
    UINT8     Capabilities[1];
} EFI_80211_ELEMENT_EXT_CAP;

Hdr Common header of an element.
Capabilities
Indicates the capabilities being advertised by the STA transmitting
the element. This is a bit field with variable length. Refer to IEEE
802.11 specification for bit value.

//**********************************************
// EFI_80211_BSS_DESP_PILOT
//**********************************************
typedef struct {
    EFI_80211_MAC_ADDRESS BSSID;
    EFI_80211_BSS_TYPE BSSType;
    UINT8 ConCapInfo;
    UINT8 ConCountryStr[2];
    UINT8 OperatingClass;
    UINT8 Channel;
    UINT8 Interval;
    EFI_80211_MULTIPLE_BSSID *MultipleBSSID;
    UINT8 RCPIMeasurement;
    UINT8 RSNIMeasurement;
} EFI_80211_BSS_DESP_PILOT;

    BSSID      Indicates a specific BSSID of the found BSS.
    BSSType    Specifies the type of the found BSS.
    ConCapInfo One octet field to report condensed capability information.
    ConCountryStr Two octet’s field to report condensed country string.
    OperatingClass Indicates the operating class value for the operating channel.
    Channel Indicates the operating channel.
    Interval Indicates the measurement pilot interval in TU.
    MultipleBSSID Indicates that the BSS is within a multiple BSSID set.
    RCPIMeasurement Specifies the RCPI of the received frame.
    RSNIMeasurement Specifies the RSNI of the received frame.

//**********************************************
// EFI_80211_MULTIPLE_BSSID
//**********************************************
typedef struct {
    EFI_80211_ELEMENT_HEADER Hdr;
    UINT8 Indicator;
    EFI_80211_SUBELEMENT_INFO SubElement[1];
} EFI_80211_MULTIPLE_BSSID;

    Hdr        Common header of an element.
    Indicator  Indicates the maximum number of BSSIDs in the multiple BSSID set.
                When Indicator is set to n, 2^n is the maximum number.
    SubElement Contains zero or more sub-elements. Type
                EFI_80211_SUBELEMENT_INFO is defined below.
typedef struct {
  UINT8 SubElementID;
  UINT8 Length;
  UINT8 Data[1];
} EFI_80211_SUBELEMENT_INFO;

SubElementID Indicates the unique identifier within the containing element or sub-element.
Length Specifies the number of octets in the Data field.
Data A variable length data buffer.

Status Codes Returned

<table>
<thead>
<tr>
<th>Code</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>EFI_SUCCESS</td>
<td>The operation completed successfully.</td>
</tr>
<tr>
<td>EFI_INVALID_PARAMETER</td>
<td>One or more of the following conditions is TRUE:</td>
</tr>
<tr>
<td></td>
<td>• This is NULL.</td>
</tr>
<tr>
<td></td>
<td>• Data is NULL.</td>
</tr>
<tr>
<td></td>
<td>• Data-&gt;Data is NULL.</td>
</tr>
<tr>
<td>EFI_UNSUPPORTED</td>
<td>One or more of the input parameters are not supported by this implementation.</td>
</tr>
<tr>
<td>EFI_ALREADY_STARTED</td>
<td>The scan operation is already started.</td>
</tr>
</tbody>
</table>

EFI_WIRELESS_MAC_CONNECTION_PROTOCOL.Associate()

Summary
Request an association with a specified peer MAC entity that is within an AP.

Prototype
typedef EFI_STATUS (EFIAPI *EFI_WIRELESS_MAC_CONNECTION_ASSOCIATE)(
  IN EFI_WIRELESS_MAC_CONNECTION_PROTOCOL *This,
  IN EFI_80211_ASSOCIATE_DATA_TOKEN *Data
);

Parameters

This Pointer to the EFI_WIRELESS_MAC_CONNECTION_PROTOCOL instance.

Data Pointer to the association token. Type EFI_80211_ASSOCIATE_DATA_TOKEN is defined in Related Definitions below.
Description

The \texttt{Associate()} function provides the capability for MAC layer to become associated with an AP.

Related Definitions

```c
//**********************************************************
// EFI_80211_ASSOCIATE_DATA_TOKEN
//**********************************************************
typedef struct {
    EFI_EVENT Event;
    EFI_STATUS Status;
    EFI_80211_ASSOCIATE_DATA *Data;
    EFI_80211_ASSOCIATE_RESULT_CODE ResultCode;
    EFI_80211_ASSOCIATE_RESULT *Result;
} EFI_80211_ASSOCIATE_DATA_TOKEN;
```

- **Event**
  This \texttt{Event} will be signaled after the \texttt{Status} field is updated by the EFI Wireless MAC Connection Protocol driver. The type of \texttt{Event} must be \texttt{EFI_NOTIFY_SIGNAL}.

- **Status**
  Will be set to one of the following values:
  - \texttt{EFI_SUCCESS}: Association operation completed successfully.
  - \texttt{EFIDEVICE_ERROR}: An unexpected network or system error occurred.

- **Data**
  Pointer to the association data. Type \texttt{EFI_80211_ASSOCIATE_DATA} is defined below.

- **ResultCode**
  Indicates the association state. Type \texttt{EFI_80211_ASSOCIATE_RESULT_CODE} is defined below.

- **Result**
  Indicates the association result. It is caller’s responsibility to free this buffer. Type \texttt{EFI_80211_ASSOCIATE_RESULT} is defined below.

The \texttt{EFI_80211_ASSOCIATE_DATA_TOKEN} structure is defined to support the process of association with a specified AP. As input, the \texttt{Data} field must be filled in by the caller of EFI Wireless MAC Connection Protocol. After the association operation completes, the EFI Wireless MAC Connection Protocol driver updates the \texttt{Status}, \texttt{ResultCode} and \texttt{Result} field and the \texttt{Event} is signaled.
typedef struct {
    EFI_80211_MAC_ADDRESS   BSSID;
    UINT16                  CapabilityInfo;
    UINT32                  FailureTimeout;
    UINT32                  ListenInterval;
    EFI_80211_ELEMENT_SUPP_CHANNEL *Channels;
    EFI_80211_ELEMENT_RSN    RSN;
    EFI_80211_ELEMENT_EXT_CAP *ExtCapElement;
    UINT8                    *VendorSpecificInfo;
} EFI_80211_ASSOCIATE_DATA;

BSSID       Specifies the address of the peer MAC entity to associate with.
CapabilityInfo  Specifies the requested operational capabilities to the AP in 2 octets.
FailureTimeout   Specifies a time limit in TU, after which the associate procedure is
terminated.
ListenInterval  Specifies if in power save mode, how often the STA awakes and
listens for the next beacon frame in TU.
Channels       Indicates a list of channels in which the STA is capable of operating.
Type EFI_80211_ELEMENT_SUPP_CHANNEL is defined below.
RSN            The cipher suites and AKM suites selected by the STA.
ExtCapElement  Specifies the parameters within the Extended Capabilities element
that are supported by the MAC entity. This is an optional parameter
and may be NULL.
VendorSpecificInfo Specifies zero or more elements. This is an optional parameter and
may be NULL.

typedef struct {
    EFI_80211_ELEMENT_HEADER   Hdr;
    EFI_80211_ELEMENT_SUPP_CHANNEL_TUPLE Subband[1];
} EFI_80211_ELEMENT_SUPP_CHANNEL;

Hdr          Common header of an element.
Subband      Indicates one or more tuples of (first channel, number of channels).
Type EFI_80211_ELEMENT_SUPP_CHANNEL_TUPLE is defined below.
/**********************************************
// EFI_80211_ELEMENT_SUPP_CHANNEL_TUPLE
//**********************************************
typedef struct {
    UINT8    FirstChannelNumber;
    UINT8    NumberOfChannels;
} EFI_80211_ELEMENT_SUPP_CHANNEL_TUPLE;

FirstChannelNumber The first channel number in a subband of supported channels.
NumberOfChannels    The number of channels in a subband of supported channels.

//**********************************************
// EFI_80211_ASSOCIATE_RESULT_CODE
//**********************************************
typedef enum {
    AssociateSuccess,
    AssociateRefusedReasonUnspecified,
    AssociateRefusedCapsMismatch,
    AssociateRefusedExtReason,
    AssociateRefusedAPOutOfMemory,
    AssociateRefusedBasicRatesMismatch,
    AssociateRejectedEmergencyServicesNotAllowed,
    AssociateRefusedTemporarily
} EFI_80211_ASSOCIATE_RESULT_CODE;

The EFI_80211_ASSOCIATE_RESULT_CODE records the result responses to the association request, which are defined in IEEE 802.11 specification.

//**********************************************
// EFI_80211_ASSOCIATE_RESULT
//**********************************************
typedef struct {
    EFI_80211_MAC_ADDRESS   BSSID;
    UINT16                   CapabilityInfo;
    UINT16                   AssociationID;
    UINT8                    RCPIValue;
    UINT8                    RSNIValue;
    EFI_80211_ELEMENT_EXT_CAP *ExtCapElement;
    EFI_80211_ELEMENT_TIMEOUT_VAL TimeoutInterval;
    UINT8                    *VendorSpecificInfo;
} EFI_80211_ASSOCIATE_RESULT;

BSSID    Specifies the address of the peer MAC entity from which the association request was received.
CapabilityInfo  Specifies the operational capabilities advertised by the AP.
AssociationID    Specifies the association ID value assigned by the AP.
RCPIValue Indicates the measured RCPI of the corresponding association request frame. It is an optional parameter and is set to zero if unavailable.

RSNIValue Indicates the measured RSNI at the time the corresponding association request frame was received. It is an optional parameter and is set to zero if unavailable.

ExtCapElement Specifies the parameters within the Extended Capabilities element that are supported by the MAC entity. This is an optional parameter and may be NULL.

TimeoutInterval Specifies the timeout interval when the result code is AssociateRefusedTemporarily.

VendorSpecificInfo Specifies zero or more elements. This is an optional parameter and may be NULL.

/*--------------------------------------------------------*/
/* EFI_80211_ELEMENT_TIMEOUT_VAL */
/*--------------------------------------------------------*/
typedef struct {
    EFI_80211_ELEMENT_HEADER Hdr;
    UINT8 Type;
    UINT32 Value;
} EFI_80211_ELEMENT_TIMEOUT_VAL;

Hdr Common header of an element.
Type Specifies the timeout interval type.
Value Specifies the timeout interval value.

Status Codes Returned

<table>
<thead>
<tr>
<th>Code</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>EFI_SUCCESS</td>
<td>The operation completed successfully.</td>
</tr>
</tbody>
</table>
| EFI_INVALID_PARAMETER| One or more of the following conditions is TRUE:
|                      | • This is NULL.                                                            |
|                      | • Data is NULL.                                                            |
|                      | • Data->Data is NULL.                                                      |
| EFI_UNSUPPORTED      | One or more of the input parameters are not supported by this implementation.|
| EFI_ALREADY_STARTED  | The association process is already started.                                |
| EFI_NOT_READY        | Authentication is not performed before this association process.           |
| EFI_NOT_FOUND        | The specified peer MAC entity is not found.                                |
| EFI_OUT_OF_RESOURCES | Required system resources could not be allocated.                          |
EFI_WIRELESS_MAC_CONNECTION_PROTOCOL.Disassociate()

Summary
Request a disassociation with a specified peer MAC entity.

Prototype

typedef
EFI_STATUS
(EFIAPI *EFI_WIRELESS_MAC_CONNECTION_DISASSOCIATE)(
    IN EFI_WIRELESS_MAC_CONNECTION_PROTOCOL *This,
    IN EFI_80211_DISASSOCIATE_DATA_TOKEN *Data
);

Parameters

This Pointer to the EFI_WIRELESS_MAC_CONNECTION_PROTOCOL instance.

Data Pointer to the disassociation token. Type EFI_80211_DISASSOCIATE_DATA_TOKEN is defined in Related Definitions below.

Description
The Disassociate() function is invoked to terminate an existing association. Disassociation is a notification and cannot be refused by the receiving peer except when management frame protection is negotiated and the message integrity check fails.

Related Definitions

//***************************************************************
// EFI_80211_DISASSOCIATE_DATA_TOKEN
//***************************************************************
typedef struct {
    EFI_EVENT Event;
    EFI_STATUS Status;
    EFI_80211_DISASSOCIATE_DATA *Data;
    EFI_80211_DISASSOCIATE_RESULT_CODE ResultCode;
} EFI_80211_DISASSOCIATE_DATA_TOKEN;

Event This Event will be signaled after the Status field is updated by the EFI Wireless MAC Connection Protocol driver. The type of Event must be EFI_NOTIFY_SIGNAL.

Status Will be set to one of the following values:

EFI_SUCCESS: Disassociation operation completed successfully.

EFI_DEVICE_ERROR: An unexpected network or system error occurred.

EFI_ACCESS_DENIED: The disassociation operation is not completed due to some underlying hardware or software state.
The disassociation operation is started but not yet completed.

Data

Pointer to the disassociation data. Type 
EFI_80211_DISASSOCIATE_DATA is defined below.

ResultCode

Indicates the disassociation state. Type 
EFI_80211_DISASSOCIATE_RESULT_CODE is defined below.

---

EFI_80211_DISASSOCIATE_DATA

typedef struct {
    EFI_80211_MAC_ADDRESS BSSID;
    EFI_80211_REASON_CODE ReasonCode;
    UINT8 *VendorSpecificInfo;
} EFI_80211_DISASSOCIATE_DATA;

BSSID

Specifies the address of the peer MAC entity with which to perform
the disassociation process.

ReasonCode

Specifies the reason for initiating the disassociation process.

VendorSpecificInfo

Zero or more elements, may be NULL.

---

EFI_80211_REASON_CODE

typedef enum {
    Ieee80211UnspecifiedReason      = 1,
    Ieee80211PreviousAuthenticateInvalid = 2,
    Ieee80211DeauthenticatedSinceLeaving = 3,
    Ieee80211DisassociatedDueToInactive = 4,
    Ieee80211DisassociatedSinceApUnable = 5,
    Ieee80211Class2FrameNonauthenticated = 6,
    Ieee80211Class3FrameNonassociated = 7,
    Ieee80211DisassociatedSinceLeaving = 8,
    // ...
} EFI_80211_REASON_CODE;

---

EFI_80211_DISASSOCIATE_RESULT_CODE

typedef enum {
    DisassociateSuccess,
    DisassociateInvalidParameters
} EFI_80211_DISASSOCIATE_RESULT_CODE;

DisassociateSuccess

Disassociation process completed successfully.

DisassociateInvalidParameters

---

Note: The reason codes are defined in chapter 8.4.1.7 Reason Code field, IEEE 802.11-2012.
Disassociation failed due to any input parameter is invalid.

**Status Codes Returned**

<table>
<thead>
<tr>
<th>Code</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>EFI_SUCCESS</td>
<td>The operation completed successfully.</td>
</tr>
<tr>
<td>EFI_INVALID_PARAMETER</td>
<td>One or more of the following conditions is TRUE:</td>
</tr>
<tr>
<td></td>
<td>• This is NULL.</td>
</tr>
<tr>
<td></td>
<td>• Data is NULL.</td>
</tr>
<tr>
<td>EFI_ALREADY_STARTED</td>
<td>The disassociation process is already started.</td>
</tr>
<tr>
<td>EFI_NOT_READY</td>
<td>The disassociation service is invoked to a nonexistent association relationship.</td>
</tr>
<tr>
<td>EFI_NOT_FOUND</td>
<td>The specified peer MAC entity is not found.</td>
</tr>
<tr>
<td>EFI_OUT_OF_RESOURCES</td>
<td>Required system resources could not be allocated.</td>
</tr>
</tbody>
</table>

**EFI_WIRELESS_MAC_CONNECTION_PROTOCOL.Authenticate()**

**Summary**

Request the process of establishing an authentication relationship with a peer MAC entity.

**Prototype**

```c
typedef EFI_STATUS
    (EFIAPI *EFI_WIRELESS_MAC_CONNECTION_AUTHENTICATE)(
    IN EFI_WIRELESS_MAC_CONNECTION_PROTOCOL *This,
    IN EFI_80211_AUTHENTICATE_DATA_TOKEN *Data
    );
```

**Parameters**

- **This**  
  Pointer to the `EFI_WIRELESS_MAC_CONNECTION_PROTOCOL` instance.

- **Data**  
  Pointer to the authentication token. Type `EFI_80211_AUTHENTICATE_DATA_TOKEN` is defined in Related Definitions below.

**Description**

The `Authenticate()` function requests authentication with a specified peer MAC entity. This service might be time-consuming thus is designed to be invoked independently of the association service.
Related Definitions

```c
typedef struct {
    EFI_EVENT Event;
    EFI_STATUS Status;
    EFI_80211_AUTHENTICATE_DATA *Data;
    EFI_80211_AUTHENTICATE_RESULT_CODE ResultCode;
    EFI_80211_AUTHENTICATE_RESULT *Result;
} EFI_80211_AUTHENTICATE_DATA_TOKEN;
```

**Event**  
This `Event` will be signaled after the `Status` field is updated by the EFI Wireless MAC Connection Protocol driver. The type of `Event` must be `EFI_NOTIFY_SIGNAL`.

**Status**  
Will be set to one of the following values:

- **EFI_SUCCESS**: Authentication operation completed successfully.
- **EFI_PROTOCOL_ERROR**: Peer MAC entity rejects the authentication.
- **EFI_NO_RESPONSE**: Peer MAC entity does not respond the authentication request.
- **EFI_DEVICE_ERROR**: An unexpected network or system error occurred.
- **EFI_ACCESS_DENIED**: The authentication operation is not completed due to some underlying hardware or software state.
- **EFI_NOT_READY**: The authentication operation is started but not yet completed.

**Data**  
Pointer to the authentication data. Type `EFI_80211_AUTHENTICATE_DATA` is defined below.

**ResultCode**  
Indicates the association state. Type `EFI_80211_AUTHENTICATE_RESULT_CODE` is defined below.

**Result**  
Indicates the association result. It is caller’s responsibility to free this buffer. Type `EFI_80211_AUTHENTICATE_RESULT` is defined below.
/**********************************************
// EFI_80211_AUTHENTICATION_DATA
//**********************************************
typedef struct {
  EFI_80211_MAC_ADDRESS BSSID;
  EFI_80211_AUTHENTICATION_TYPE AuthType;
  UINT32 FailureTimeout;
  UINT8 *FTContent;
  UINT8 *SAEContent;
  UINT8 *VendorSpecificInfo;
} EFI_80211_AUTHENTICATE_DATA;

BSSID Specifies the address of the peer MAC entity with which to perform the authentication process.

AuthType Specifies the type of authentication algorithm to use during the authentication process.

FailureTimeout Specifies a time limit in TU after which the authentication procedure is terminated.

FTContent Specifies the set of elements to be included in the first message of the FT authentication sequence, may be NULL.

SAEContent Specifies the set of elements to be included in the SAE Commit Message or SAE Confirm Message, may be NULL.

VendorSpecificInfo Zero or more elements, may be NULL.

//**********************************************
// EFI_80211_AUTHENTICATION_TYPE
//**********************************************
typedef enum {
  OpenSystem,
  SharedKey,
  FastBSSTransition,
  SAE
} EFI_80211_AUTHENTICATION_TYPE;

OpenSystem Open system authentication, admits any STA to the DS.

SharedKey Shared Key authentication relies on WEP to demonstrate knowledge of a WEP encryption key.

FastBSSTransition FT authentication relies on keys derived during the initial mobility domain association to authenticate the stations.

SAE SAE authentication uses finite field cryptography to prove knowledge of a shared password.
The result code indicates the result response to the authentication request from the peer MAC entity.

```c
typedef enum {
    AuthenticateSuccess,
    AuthenticateRefused,
    AuthenticateAnticLoggingTokenRequired,
    AuthenticateFiniteCyclicGroupNotSupported,
    AuthenticationRejected,
    AuthenticateInvalidParameter
} EFI_80211_AUTHENTICATE_RESULT_CODE;
```

Status Codes Returned

<table>
<thead>
<tr>
<th>Code</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>EFI_SUCCESS</td>
<td>The operation completed successfully.</td>
</tr>
<tr>
<td>EFI_INVALID_PARAMETER</td>
<td>One or more of the following conditions is TRUE:</td>
</tr>
<tr>
<td></td>
<td>• This is NULL.</td>
</tr>
<tr>
<td></td>
<td>• Data is NULL.</td>
</tr>
<tr>
<td></td>
<td>• Data.Data is NULL.</td>
</tr>
<tr>
<td>EFI_UNSUPPORTED</td>
<td>One or more of the input parameters are not supported by this implementation.</td>
</tr>
<tr>
<td>EFI_ALREADY_STARTED</td>
<td>The authentication process is already started.</td>
</tr>
<tr>
<td>EFI_NOT_FOUND</td>
<td>The specified peer MAC entity is not found.</td>
</tr>
<tr>
<td>EFI_OUT_OF_RESOURCES</td>
<td>Required system resources could not be allocated.</td>
</tr>
</tbody>
</table>
**EFI_WIRELESS_MAC_CONNECTION_PROTOCOL.Deauthenticate()**

**Summary**
Invalidate the authentication relationship with a peer MAC entity.

**Prototype**
```c
typedef EFI_STATUS (EFIAPI *EFI_WIRELESS_MAC_CONNECTION_DEAUTHENTICATE)(
    IN EFI_WIRELESS_MAC_CONNECTION_PROTOCOL *This,
    IN EFI_80211_DEAUTHENTICATE_DATA_TOKEN *Data
);
```

**Parameters**
- **This**
  Pointer to the **EFI_WIRELESS_MAC_CONNECTION_PROTOCOL** instance.
- **Data**
  Pointer to the deauthentication token. Type **EFI_80211_DEAUTHENTICATE_DATA_TOKEN** is defined in Related Definitions below.

**Description**
The **Deauthenticate()** function requests that the authentication relationship with a specified peer MAC entity be invalidated. Deauthentication is a notification and when it is sent out the association at the transmitting station is terminated.

**Related Definitions**
```c
//****************************************************************************
// EFI_80211_DEAUTHENTICATE_DATA_TOKEN
//****************************************************************************
typedef struct {
    EFI_EVENT Event;
    EFI_STATUS Status;
    EFI_80211_DEAUTHENTICATE_DATA *Data;
} EFI_80211_DEAUTHENTICATE_DATA_TOKEN;
```

- **Event**
  This **Event** will be signaled after the **Status** field is updated by the EFI Wireless MAC Connection Protocol driver. The type of **Event** must be **EFI_NOTIFY_SIGNAL**.

- **Status**
  Will be set to one of the following values:
  - **EFI_SUCCESS**: Deauthentication operation completed successfully.
  - **EFI_DEVICE_ERROR**: An unexpected network or system error occurred.
  - **EFI_ACCESS_DENIED**: The deauthentication operation is not completed due to some underlying hardware or software state.
**EFI_NOT_READY**: The deauthentication operation is started but not yet completed.

**Data**: Pointer to the deauthentication data. Type **EFI_80211_DEAUTHENTICATE_DATA** is defined below.

```c
//***************************************************************************
// EFI_80211_DEAUTHENTICATE_DATA
//***************************************************************************
typedef struct {
   EFI_80211_MAC_ADDRESS  BSSID;
   EFI_80211_REASON_CODE  ReasonCode;
   UINT8                  *VendorSpecificInfo;
} EFI_80211_DEAUTHENTICATE_DATA;
```

**BSSID**: Specifies the address of the peer MAC entity with which to perform the deauthentication process.

**ReasonCode**: Specifies the reason for initiating the deauthentication process.

**VendorSpecificInfo**: Zero or more elements, may be NULL.

### Status Codes Returned

<table>
<thead>
<tr>
<th>Status Code</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>EFI_SUCCESS</td>
<td>The operation completed successfully.</td>
</tr>
<tr>
<td>EFI_INVALID_PARAMETER</td>
<td>One or more of the following conditions is TRUE:</td>
</tr>
<tr>
<td></td>
<td>• <strong>This</strong> is NULL.</td>
</tr>
<tr>
<td></td>
<td>• <strong>Data</strong> is NULL.</td>
</tr>
<tr>
<td></td>
<td>• <strong>Data&gt;Data</strong> is NULL.</td>
</tr>
<tr>
<td>EFI_ALREADY_STARTED</td>
<td>The deauthentication process is already started.</td>
</tr>
<tr>
<td>EFI_NOT_READY</td>
<td>The deauthentication service is invoked to a nonexistent association or authentication relationship.</td>
</tr>
<tr>
<td>EFI_NOT_FOUND</td>
<td>The specified peer MAC entity is not found.</td>
</tr>
<tr>
<td>EFI_OUT_OF_RESOURCES</td>
<td>Required system resources could not be allocated.</td>
</tr>
</tbody>
</table>

### 27.4 EFI Wireless MAC Connection II Protocol

This section provides a detailed description of EFI Wireless MAC Connection II Protocol.

**EFI_WIRELESS_MAC_CONNECTION_II_PROTOCOL**

**Summary**

The **EFI_WIRELESS_MAC_CONNECTION_II_PROTOCOL** provides network management service interfaces for 802.11 network stack. It is used by network applications (and drivers) to establish wireless connection with a wireless network.
GUID

```c
#define EFI_WIRELESS_MAC_CONNECTION_II_PROTOCOL_GUID {
    0xb0f9bf, 0x699d, 0x4fdd, 
    { 0xa7, 0xc3, 0x25, 0x46, 0x68, 0x1b, 0xf6, 0x3b }}
```

Protocol Interface Structure

```c
typedef struct _EFI_WIRELESS_MAC_CONNECTION_II_PROTOCOL {
    EFI_WIRELESS_MAC_CONNECTION_II_GET_NETWORKS GetNetworks;
    EFI_WIRELESS_MAC_CONNECTION_II_CONNECT_NETWORK ConnectNetwork;
    EFI_WIRELESS_MAC_CONNECTION_II_DISCONNECT_NETWORK DisconnectNetwork;
} EFI_WIRELESS_MAC_CONNECTION_II_PROTOCOL;
```

Parameters

- **GetNetworks**: Get a list of nearby detectable wireless network. See the `GetNetworks()` function description.
- **ConnectNetwork**: Places a connection request with a specific wireless network. See the `ConnectNetwork()` function description.
- **DisconnectNetwork**: Places a disconnection request with a specific wireless network. See the `DisconnectNetwork()` function description.

Description

The **EFI_WIRELESS_MAC_CONNECTION_II_PROTOCOL** is designed to provide management service interfaces for the EFI wireless network stack to establish relationship with a wireless network (identified by **EFI_80211_NETWORK** defined below). An EFI Wireless MAC Connection II Protocol instance will be installed on each communication device that the EFI wireless network stack runs on.

**EFI_WIRELESS_MAC_CONNECTION_II_PROTOCOL.GetNetworks()**

**Summary**

Request a survey of potential wireless networks that administrator can later elect to try to join.

**Prototype**

```c
typedef
EFI_STATUS
(EIFIAPI *EFI_WIRELESS_MAC_CONNECTION_II_GET_NETWORKS)(
    IN EFI_WIRELESS_MAC_CONNECTION_II_PROTOCOL *This,
    IN EFI_80211_GET_NETWORKS_TOKEN *Token
);
```

**Parameters**

- **This**: Pointer to the **EFI_WIRELESS_MAC_CONNECTION_II_PROTOCOL** instance.
- **Token**: Pointer to the token for getting wireless network. Type **EFI_80211_GET_NETWORKS_TOKEN** is defined in Related Definitions below.
Description
The `GetNetworks()` function returns the description of a list of wireless networks detected by wireless UNDI driver. This function is always non-blocking. If the operation succeeds or fails due to any error, the `Token->Event` will be signaled and `Token->Status` will be updated accordingly. The caller of this function is responsible for inputting SSIDs in case of searching hidden networks.

Related Definitions

```c
typedef struct {
    EFI_EVENT Event;
    EFI_STATUS Status;
    EFI_80211_GET_NETWORKS_DATA *Data;
    EFI_80211_GET_NETWORKS_RESULT *Result;
} EFI_80211_GET_NETWORKS_TOKEN;
```

- **Event**: If the status code returned by `GetNetworks()` is `EFI_SUCCESS`, then this Event will be signaled after the `Status` field is updated by the EFI Wireless MAC Connection Protocol II driver. The type of `Event` must be `EFI_NOTIFY_SIGNAL`.

- **Status**: Will be set to one of the following values:
  - `EFI_SUCCESS`: The operation completed successfully.
  - `EFI_NOT_FOUND`: Failed to find available wireless networks.
  - `EFI_DEVICE_ERROR`: An unexpected network or system error occurred.
  - `EFI_ACCESS_DENIED`: The operation is not completed due to some underlying hardware or software state.
  - `EFI_NOT_READY`: The operation is started but not yet completed.

- **Data**: Pointer to the input data for getting networks. Type `EFI_80211_GET_NETWORKS_DATA` is defined below.

- **Result**: Indicates the scan result. It is caller's responsibility to free this buffer. Type `EFI_80211_GET_NETWORKS_RESULT` is defined below.

```c
typedef struct {
    UINT32 NumOfSSID;
    EFI_80211_SSID SSIDLList[1];
} EFI_80211_GET_NETWORKS_DATA;
```
NumOfSSID  The number of EFI_80211_SSID in SSIDList. If zero, SSIDList should be ignored.

SSIDList  The SSIDList is a pointer to an array of EFI_80211_SSID instances. The number of entries is specified by NumOfSSID. The array should only include SSIDs of hidden networks. It is suggested that the caller inputs less than 10 elements in the SSIDList. It is the caller's responsibility to free this buffer. Type EFI_80211_SSID is defined below.

#define EFI_MAX_SSID_LEN 32

//**********************************************
// EFI_80211_SSID
//**********************************************
typedef struct {
    UINT8 SSIdLen;
    UINT8 SSId[EFI_MAX_SSID_LEN];
} EFI_80211_SSID;

SSIdLen  Length in bytes of the SSId. If zero, ignore SSId field.

SSID  Specifies the service set identifier.

//**********************************************
// EFI_80211_GET_NETWORKS_RESULT
//**********************************************
typedef struct {
    UINT8 NumOfNetworkDesc;
    EFI_80211_NETWORK_DESCRIPTION NetworkDesc[1];
} EFI_80211_GET_NETWORKS_RESULT;

NumOfNetworkDesc  The number of elements in NetworkDesc. If zero, NetworkDesc should be ignored.

NetworkDesc  The NetworkDesc is a variable-length array of elements of type EFI_80211_NETWORK_DESCRIPTION. Type EFI_80211_NETWORK_DESCRIPTION is defined below.

//**********************************************
// EFI_80211_NETWORK_DESCRIPTION
//**********************************************
typedef struct {
    EFI_80211_NETWORK Network;
    UINT8 NetworkQuality;
} EFI_80211_NETWORK_DESCRIPTION;

Network  Specifies the found wireless network. Type EFI_80211_NETWORK is defined below.

NetworkQuality  Indicates the network quality as a value between 0 to 100, where 100 indicates the highest network quality.
typedef struct {
  EFI_80211_BSS_TYPE BSSType;
  EFI_80211_SSID SSId;
  EFI_80211_AKM_SUITE_SELECTOR *AKMSuite;
  EFI_80211_CIPHER_SUITE_SELECTOR *CipherSuite;
} EFI_80211_NETWORK;

BSSType Specifies the type of the BSS. Type EFI_80211_BSS_TYPE is defined below.

SSId Specifies the SSID of the BSS. Type EFI_80211_SSID is defined above.

AKMSuite Pointer to the AKM suites supported in the wireless network. Type EFI_80211_AKM_SUITE_SELECTOR is defined below.

CipherSuite Pointer to the cipher suites supported in the wireless network. Type EFI_80211_CIPHER_SUITE_SELECTOR is defined below.

typedef enum {
  IeeeInfrastructureBSS,  
  IeeeIndependentBSS, 
  IeeeMeshBSS,           
  IeeeAnyBss            
} EFI_80211_BSS_TYPE;

The EFI_80211_BSS_TYPE is defined to enumerate BSS type.

typedef struct {
  UINT8 Oui[3];
  UINT8 SuiteType;
} EFI_80211_SUITE_SELECTOR;

Oui Organization Unique Identifier, as defined in IEEE 802.11 standard, usually set to 00-0F-AC.

SuiteType Suites types, as defined in IEEE 802.11 standard.
//EFI_80211_AKM_SUITE_SELECTOR
typedef struct {
    UINT16 AKMSuiteCount;
    EFI_80211_SUITE_SELECTOR AKMSuiteList[1];
} EFI_80211_AKM_SUITE_SELECTOR;

AKMSuiteCount Indicates the number of AKM suite selectors that are contained in AKMSuiteList. If zero, the AKMSuiteList is ignored.

AKMSuiteList A variable-length array of AKM suites, as defined in IEEE 802.11 standard, Table 8-101. The number of entries is specified by AKMSuiteCount.

//EFI_80211_CIPHER_SUITE_SELECTOR
typedef struct {
    UINT16 CipherSuiteCount;
    EFI_80211_SUITE_SELECTOR CipherSuiteList[1];
} EFI_80211_CIPHER_SUITE_SELECTOR;

CipherSuiteCount Indicates the number of cipher suites that are contained in CipherSuiteList. If zero, the CipherSuiteList is ignored.

CipherSuiteList A variable-length array of cipher suites, as defined in IEEE 802.11 standard, Table 8-99. The number of entries is specified by CipherSuiteCount.

Status Codes Returned

<table>
<thead>
<tr>
<th>Code</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>EFI_SUCCESS</td>
<td>The operation started, and an event will eventually be raised for the caller.</td>
</tr>
<tr>
<td>EFI_INVALID_PARAMETER</td>
<td>One or more of the following conditions is TRUE: This is NULL. Token is NULL.</td>
</tr>
<tr>
<td>EFI_UNSUPPORTED</td>
<td>One or more of the input parameters is not supported by this implementation.</td>
</tr>
<tr>
<td>EFI_ALREADY_STARTED</td>
<td>The operation of getting wireless network is already started.</td>
</tr>
<tr>
<td>EFI_OUT_OF_RESOURCES</td>
<td>Required system resources could not be allocated.</td>
</tr>
</tbody>
</table>

EFI_WIRELESS_MAC_CONNECTION_II_PROTOCOL.ConnectNetwork()

Summary
Connect a wireless network specified by a particular SSID, BSS type and Security type.
Prototype

typedef
  EFI_STATUS
  (EFIAPI *EFI_WIRELESS_MAC_CONNECTION_II_CONNECT_NETWORK)(
   IN EFI_WIRELESS_MAC_CONNECTION_II_PROTOCOL *This,
   IN EFI_80211_CONNECT_NETWORK_TOKEN *Token
  );

Parameters

  This  Pointer to the EFI_WIRELESS_MAC_CONNECTION_II_PROTOCOL instance.
  Token Pointer to the token for connecting wireless network. Type EFI_80211_CONNECT_NETWORK_TOKEN is defined in Related Definitions below.

Description

The ConnectNetwork() function places a request to wireless UNDI driver to connect a wireless network specified by a particular SSID, BSS type, Authentication method and cipher. This function will trigger wireless UNDI driver to perform authentication and association process to establish connection with a particular Access Point for the specified network. This function is always non-blocking. If the connection succeeds or fails due to any error, the Token->Event will be signaled and Token->Status will be updated accordingly.

After having signaled a successful connection completion, the UNDI driver will update the network connection state using the network media state information type in the EFI_ADAPTER_INFORMATION_PROTOCOL. If needed, the caller should use EFI_ADAPTER_INFORMATION_PROTOCOL to regularly get the network media state to find if the UNDI driver is still connected to the wireless network (EFI_SUCCESS) or not (EFI_NO_MEDIA).

Generally a driver or application in WiFi stack would provide user interface to end user to manage profiles for selecting which wireless network to join and other state management. This module should prompt the user to select a network and input WiFi security data such as certificate, private key file, password, etc. Then the module should deploy WiFi security data through EFI Supplicant Protocol and/ or EFI EAP Configuration Protocol before calling ConnectNetwork() function.

Related Definitions

    /***************************************************************************/
    // EFI_80211_CONNECT_NETWORK_TOKEN
    /***************************************************************************/
    typedef struct {
      EFI_EVENT Event;
      EFI_STATUS Status;
      EFI_80211_CONNECT_NETWORK_DATA *Data;
      EFI_80211_CONNECT_NETWORK_RESULT_CODE ResultCode;
    } EFI_80211_CONNECT_NETWORK_TOKEN;
If the status code returned by `ConnectNetwork()` is `EFI_SUCCESS`, then this `Event` will be signaled after the `Status` field is updated by the EFI Wireless MAC Connection Protocol II driver. The type of `Event` must be `EFI_NOTIFY_SIGNAL`.

The `EFI_80211_CONNECT_NETWORK_TOKEN` structure is defined to support the process of determining the characteristics of the available networks. As input, the `Data` field must be filled in by the caller of EFI Wireless MAC Connection II Protocol. After the operation completes, the EFI Wireless MAC Connection II Protocol driver updates the `Status` and `ResultCode` field and the `Event` is signaled.

```c
typedef struct {
  EFI_80211_NETWORK *Network;
  UINT32 FailureTimeout;
} EFI_80211_CONNECT_NETWORK_DATA;
```

- **Network** Specifies the wireless network to connect to. Type `EFI_80211_NETWORK` is defined above.
- **FailureTimeout** Specifies a time limit in seconds that is optionally present, after which the connection establishment procedure is terminated by the UNDI driver. This is an optional parameter and may be 0. Values of 5 seconds or higher are recommended.
typedef enum {
  ConnectSuccess,
  ConnectRefused,
  ConnectFailed,
  ConnectFailureTimeout,
  ConnectFailedReasonUnspecified
} EFI_80211_CONNECT_NETWORK_RESULT_CODE;

ConnectSuccess      The connection establishment operation finished successfully.
ConnectRefused      The connection was refused by the Network.
ConnectFailed       The connection establishment operation failed (i.e, Network is not detected).
ConnectFailureTimeout
                        The connection establishment operation was terminated on timeout.
ConnectFailedReasonUnspecified
                        The connection establishment operation failed on other reason.

Status Codes Returned

<table>
<thead>
<tr>
<th>Code</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>EFI_SUCCESS</td>
<td>The operation started successfully. Results will be notified eventually.</td>
</tr>
<tr>
<td>EFI_INVALID_PARAMETER</td>
<td>One or more of the following conditions is TRUE:</td>
</tr>
<tr>
<td></td>
<td>This is NULL.</td>
</tr>
<tr>
<td></td>
<td>Token is NULL.</td>
</tr>
<tr>
<td>EFI_UNSUPPORTED</td>
<td>One or more of the input parameters are not supported by this implementation.</td>
</tr>
<tr>
<td>EFI_ALREADY_STARTED</td>
<td>The connection process is already started.</td>
</tr>
<tr>
<td>EFI_NOT_FOUND</td>
<td>The specified wireless network is not found.</td>
</tr>
<tr>
<td>EFI_OUT_OF_RESOURCES</td>
<td>Required system resources could not be allocated.</td>
</tr>
</tbody>
</table>
**EFI_WIRELESS_MAC_CONNECTION_II_PROTOCOL.DisconnectNetwork()**

**Summary**
Request a disconnection with current connected wireless network.

**Prototype**

typedef
EFI_STATUS
(EIFIAPI *EFI_WIRELESS_MAC_CONNECTION_II_DISCONNECT_NETWORK)(
   IN EFI_WIRELESS_MAC_CONNECTION_II_PROTOCOL *This,
   IN EFI_80211_DISCONNECT_NETWORK_TOKEN   *Token
);

**Parameters**

- **This** Pointer to the EFI_WIRELESS_MAC_CONNECTION_II_PROTOCOL instance.
- **Token** Pointer to the token for disconnecting wireless network. Type EFI_80211_DISCONNECT_NETWORK_TOKEN is defined in Related Definitions below.

**Description**
The DisconnectNetwork() function places a request to wireless UNDI driver to disconnect from the wireless network it is connected to. This function will trigger the wireless UNDI driver to perform disassociation and deauthentication process to terminate an existing connection. This function is always non-blocking. After wireless UNDI driver received acknowledgment frame from AP and freed up corresponding resources, the Token->Event will be signaled and Token->Status will be updated accordingly.

**Related Definitions**

```c
//******************************************************************************
// E80211_DISCONNECT_NETWORK_TOKEN //
******************************************************************************
typedef struct {
   EFI_EVENT Event;
   EFI_STATUS Status;
} EFI_80211_DISCONNECT_NETWORK_TOKEN;
```

- **Event** If the status code returned by DisconnectNetwork() is EFI_SUCCESS, then this Event will be signaled after the Status field is updated by the EFI Wireless MAC Connection Protocol II driver. The type of Event must be EFI_NOTIFY_SIGNAL.
- **Status** Will be set to one of the following values:
  - **EFI_SUCCESS**: The operation completed successfully
  - **EFI_DEVICE_ERROR**: An unexpected network or system error occurred.
**EFI_ACCESS_DENIED:** The operation is not completed due to some underlying hardware or software state.

### Status Codes Returned

<table>
<thead>
<tr>
<th>Code</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>EFI_SUCCESS</td>
<td>The operation started successfully. Results will be notified eventually.</td>
</tr>
<tr>
<td>EFI_INVALID_PARAMETER</td>
<td>One or more of the following conditions is TRUE:</td>
</tr>
<tr>
<td></td>
<td>This is NULL.</td>
</tr>
<tr>
<td></td>
<td>Token is NULL.</td>
</tr>
<tr>
<td>EFI_UNSUPPORTED</td>
<td>One or more of the input parameters are not supported by this implementation.</td>
</tr>
<tr>
<td>EFI_NOT_FOUND</td>
<td>Not connected to a wireless network.</td>
</tr>
<tr>
<td>EFI_OUT_OF_RESOURCES</td>
<td>Required system resources could not be allocated.</td>
</tr>
</tbody>
</table>

### 27.5 EFI Supplicant Protocol

This section defines the EFI Supplicant Protocol.

#### 27.5.1 Supplicant Service Binding Protocol

**EFI_SUPPLICANT_SERVICE_BINDING_PROTOCOL**

**Summary**

The EFI Supplicant Service Binding Protocol is used to locate EFI Supplicant Protocol drivers to create and destroy child of the driver to communicate with other host using Supplicant protocol.

**GUID**

```c
#define EFI_SUPPLICANT_SERVICE_BINDING_PROTOCOL_GUID  
  { 0x45bcd98e, 0x59ad, 0x4174, 
  { 0x95, 0x46, 0x34, 0x4a, 0x7, 0x48, 0x58, 0x98 }}
```

**Description**

A module that requires supplicant services can call one of the protocol handler services, such as `BS-LocateHandleBuffer()`, to search devices that publish an EFI Supplicant Service Binding Protocol GUID. Such device supports the EFI Supplicant Protocol and may be available for use. After a successful call to the `EFI_SUPPLICANT_SERVICE_BINDING_PROTOCOL.CreateChild()` function, the newly created child EFI Supplicant Protocol driver is in an un-configured state; it is not ready to do any operation until configured via SetData(). Every successful call to the `EFI_SUPPLICANTSERVICE_BINDING_PROTOCOL.CreateChild()` function must be matched with a call to the `EFI_SUPPLICANT_SERVICE_BINDING_PROTOCOL.DestroyChild()` function to release the protocol driver.
27.5.2 Supplicant Protocol

**EFI_SUPPLICANT_PROTOCOL**

**Summary**

This protocol provides services to process authentication and data encryption/decryption for security management.

**GUID**

```c
#define EFI_SUPPLICANT_PROTOCOL_GUID \ 
{ 0x54fcc43e, 0xaa89, 0x4333, \ 
{ 0x9a, 0x85, 0xcd, 0xea, 0x24, 0x5, 0x1e, 0x9e }}
```

**Protocol Interface Structure**

```c
typedef struct _EFI_SUPPLICANT_PROTOCOL {
  EFI_SUPPLICANT_BUILD_RESPONSE_PACKET BuildResponsePacket;
  EFI_SUPPLICANT_PROCESS_PACKET  ProcessPacket;
  EFI_SUPPLICANT_SET_DATA       SetData;
  EFI_SUPPLICANT_GET_DATA       GetData;
} EFI_SUPPLICANT_PROTOCOL;
```

**Parameters**

- **BuildResponsePacket**: This API processes security data for handling key management. See the `BuildResponsePacket()` function description.
- **ProcessPacket**: This API processes frame for encryption or decryption. See the `ProcessPacket()` function description.
- **SetData**: This API sets the information needed during key generated in handshake. See the `SetData()` function description.
- **GetData**: This API gets the information generated in handshake. See the `GetData()` function description.

**Description**

The **EFI_SUPPLICANT_PROTOCOL** is designed to provide unified place for WIFI and EAP security management. Both PSK authentication and 802.1X EAP authentication can be managed via this protocol and driver or application as a consumer can only focus on about packet transmitting or receiving. For 802.1X EAP authentication, an instance of **EFI_EAP_CONFIGURATION_PROTOCOL** must be installed to the same handle as the EFI Suppliant Protocol.

**EFI_SUPPLICANT_PROTOCOL.BuildResponsePacket()**

**Summary**

`BuildResponsePacket()` is called during STA and AP authentication is in progress. Supplicant derives the PTK or session keys depend on type of authentication is being employed.
Prototype

```c
typedef EFI_STATUS
  (EFIAPIC *EFI_SUPPLICANT_BUILD_RESPONSE_PACKET)(
    IN EFI_SUPPLICANT_PROTOCOL *This,
    IN UINT8          *RequestBuffer,  OPTIONAL
    IN UINTN RequestBufferSize, OPTIONAL
    OUT UINT8          *Buffer,
    IN OUT UINTN        *BufferSize
  );
```

Parameters

- **This**: Pointer to the `EFI_SUPPLICANT_PROTOCOL` instance.
- **RequestBuffer**: Pointer to the most recently received EAPOL packet. `NULL` means the supplicant need initiate the EAP authentication session and send EAPOL-Start message.
- **RequestSize**: Packet size in bytes for the most recently received EAPOL packet. 0 is only valid when `RequestBuffer` is `NULL`.
- **Buffer**: Pointer to the buffer to hold the built packet.
- **BufferSize**: Pointer to the buffer size in bytes. On input, it is the buffer size provided by the caller. On output, it is the buffer size in fact needed to contain the packet.

Description

The consumer calls `BuildResponsePacket()` when it receives the security frame. It simply passes the data to supplicant to process the data. It could be WPA-PSK which starts the 4-way handshake, or WPA-EAP first starts with Authentication process and then 4-way handshake, or 2-way group key handshake. In process of authentication, 4-way handshake or group key handshake, Supplicant needs to communicate with its peer (AP/AS) to fill the output buffer parameter. Once the 4 way handshake or group key handshake is over, and PTK (Pairwise Transient keys) and GTK (Group Temporal Key) are generated.

Status Codes Returned

<table>
<thead>
<tr>
<th>Code</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>EFI_SUCCESS</td>
<td>The required EAPOL packet is built successfully.</td>
</tr>
<tr>
<td>EFI_INVALID_PARAMETER</td>
<td>One or more of the following conditions is TRUE:</td>
</tr>
<tr>
<td></td>
<td>• RequestBuffer is <code>NULL</code>, but RequestSize is NOT 0.</td>
</tr>
<tr>
<td></td>
<td>• RequestSize is 0.</td>
</tr>
<tr>
<td></td>
<td>• Buffer is <code>NULL</code>, but RequestBuffer is NOT 0.</td>
</tr>
<tr>
<td></td>
<td>• RequestSize is 0.</td>
</tr>
<tr>
<td></td>
<td>• BufferSize is <code>NULL</code></td>
</tr>
<tr>
<td>EFI_BUFFER_TOO_SMALL</td>
<td>BufferSize is too small to hold the response packet.</td>
</tr>
<tr>
<td>EFI_NOT_READY</td>
<td>Current EAPOL session state is NOT ready to build ResponsePacket.</td>
</tr>
</tbody>
</table>
EFI_SUPPLICANT_PROTOCOL.ProcessPacket()

Summary

ProcessPacket() is called to Supplicant driver to encrypt or decrypt the data depending type of authentication type.

Prototype

typedef
EFI_STATUS
(EIFIAPI *EFI_SUPPLICANT_PROCESS_PACKET)(
    IN EFI_SUPPLICANT_PROTOCOL *This,
    IN OUT EFI_SUPPLICANT_FRAGMENT_DATA **FragmentTable,
    IN UINT32 *FragmentCount,
    IN EFI_SUPPLICANT_CRYPT_MODE CryptMode
);

Parameters

This          Pointer to the EFI_SUPPLICANT_PROTOCOL instance.
FragmentTable Pointer to a list of fragment. The caller will take responsible to handle the original FragmentTable while it may be reallocated in Supplicant driver.
FragmentCount Number of fragment.
CryptMode      Crypt mode.

Description

ProcessPacket() is responsible for encrypting or decrypting the data traffic as per authentication type. The consumer routes the data frame as it is to Supplicant module and encrypts or decrypts packet with updated length comes as output parameter. Supplicant holds the derived PTK and GTKs and uses this key to encrypt or decrypt the network traffic.

If the Supplicant driver does not support any encryption and decryption algorithm, then EFI_UNSUPPORTED is returned.

Related Definitions

//******************************************************************************
// EFI_SUPPLICANT_FRAGMENT_DATA
//******************************************************************************
typedef struct {
    UINT32     FragmentLength;
    VOID       *FragmentBuffer;
} EFI_SUPPLICANT_FRAGMENT_DATA;

FragmentLength Length of data buffer in the fragment.
FragmentBuffer   Pointer to the data buffer in the fragment.
typedef enum {
    EfiSupplicantEncrypt,
    EfiSupplicantDecrypt,
} EFI_SUPPLICANT_CRYPT_MODE;

EfiSupplicantEncrypt Encrypt data provided in the fragment buffers.
EfiSupplicantDecrypt Decrypt data provided in the fragment buffers.

**Status Codes Returned**

<table>
<thead>
<tr>
<th>Code</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>EFI_SUCCESS</td>
<td>The operation completed successfully.</td>
</tr>
<tr>
<td>EFI_INVALID_PARAMETER</td>
<td>One or more of the following conditions is TRUE:</td>
</tr>
<tr>
<td></td>
<td>• FragmentTable is NULL.</td>
</tr>
<tr>
<td></td>
<td>• FragmentCount is NULL.</td>
</tr>
<tr>
<td></td>
<td>• CryptMode is invalid.</td>
</tr>
<tr>
<td>EFI_NOT_READY</td>
<td>Current supplicant state is NOT Authenticated.</td>
</tr>
<tr>
<td>EFI_ABORTED</td>
<td>Something wrong decryption the message.</td>
</tr>
<tr>
<td>EFI_UNSUPPORTED</td>
<td>This API is not supported.</td>
</tr>
</tbody>
</table>

**EFI_SUPPLICANT_PROTOCOL.SetData()**

**Summary**
Set Supplicant configuration data.

**Prototype**
```c
typedef
    EFI_STATUS
    (EFIAPI *EFI_SUPPLICANT_SET_DATA)(
        IN EFI_SUPPLICANT_PROTOCOL *This,
        IN EFI_SUPPLICANT_DATA_TYPE  DataType,
        IN VOID                      *Data,
        IN UINTN                     DataSize
    );
```

**Parameters**
- **This** Pointer to the EFI_SUPPLICANT_PROTOCOL instance.
- **DataType** The type of data.
- **Data** Pointer to the buffer to hold the data.
- **DataSize** Pointer to the buffer size in bytes.
Description

The `SetData()` function sets Supplicant configuration. For example, Supplicant driver need to know Password and TargetSSIDName to calculate PSK. Supplicant driver need to know StationMac and TargetSSIDMac to calculate PTK. Then it can derive KCK(key confirmation key) which is needed to calculate MIC, and KEK(key encryption key) which is needed to unwrap GTK.

Related Definitions

```c
typedef enum {
    EfiSupplicant80211AKMSuite,
    EfiSupplicant80211GroupDataCipherSuite,
    EfiSupplicant80211PairwiseCipherSuite,
    EfiSupplicant80211PskPassword,
    EfiSupplicant80211TargetSSIDName,
    EfiSupplicant80211StationMac,
    EfiSupplicant80211TargetSSIDMac,
    EfiSupplicant80211PTK,
    EfiSupplicant80211GTK,
    EfiSupplicantState,
    EfiSupplicant80211LinkState,
    EfiSupplicantKeyRefresh,
    EfiSupplicant80211SupportedAKMSuites,
    EfiSupplicant80211SupportedSoftwareCipherSuites,
    EfiSupplicant80211SupportedHardwareCipherSuites,
    EfiSupplicant80211IGTK,
    EfiSupplicant80211PMK,
    EfiSupplicantDataTypeMaximum
} EFI_SUPPLICANT_DATA_TYPE;
```
EfiSupplicant80211AKMSuite
Current authentication type in use. The corresponding Data is of type EFI_80211_AKM_SUITE_SELECTOR.

EfiSupplicant80211GroupDataCipherSuite
Group data encryption type in use. The corresponding Data is of type EFI_80211_CIPHER_SUITE_SELECTOR.

EfiSupplicant80211PairwiseCipherSuite
Pairwise encryption type in use. The corresponding Data is of type EFI_80211_CIPHER_SUITE_SELECTOR.

EfiSupplicant80211PskPassword
PSK password. The corresponding Data is a NULL-terminated ASCII string.

EfiSupplicant80211TargetSSIDName
Target SSID name. The corresponding Data is of type EFI_80211_SSID.

EfiSupplicant80211StationMac
Station MAC address. The corresponding Data is of type EFI_80211_MAC_ADDRESS.

EfiSupplicant80211TargetSSIDMac
Target SSID MAC address. The corresponding Data is 6 bytes MAC address.

EfiSupplicant80211PTK
802.11 PTK. The corresponding Data is of type EFI_SUPPLICANT_KEY.

EfiSupplicant80211GTK
802.11 GTK. The corresponding Data is of type EFI_SUPPLICANT_GTK_LIST.

EfiSupplicantState
Supplicant state. The corresponding Data is EFI_EAPOL_SUPPLICANT_PAE_STATE.

EfiSupplicant80211LinkState
802.11 link state. The corresponding Data is EFI_80211_LINK_STATE.

EfiSupplicantKeyRefresh
Flag indicates key is refreshed. The corresponding Data is EFI_SUPPLICANT_KEY_REFRESH.

EfiSupplicant80211SupportedAKMSuites
Supported authentication types. The corresponding Data is of type EFI_80211_AKM_SUITE_SELECTOR.

EfiSupplicant80211SupportedSoftwareCipherSuites
Supported software encryption types provided by supplicant driver.
The corresponding Data is of type 
**EFI_80211_CIPHER_SUITE_SELECTOR**.

EfiSupplicant80211SupportedHardwareCipherSuites

Supported hardware encryption types provided by wireless UNDI
driver. The corresponding Data is of type 
**EFI_80211_CIPHER_SUITE_SELECTOR**.

EfiSupplicant80211IGTK

802.11 Integrity GTK. The corresponding Data is of type 
**EFI_SUPPLICANT_GTK_LIST**.

EfiSupplicant80211IPMK

802.11 PMK. The corresponding Data is 32 bytes pairwise master
key.

',//**********************************************
// EFI_80211_LINK_STATE
//**********************************************
typedef enum {
    Ieee80211UnauthenticatedUnassociated,
    Ieee80211AuthenticatedUnassociated,
    Ieee80211PendingRSNAuthentication,
    Ieee80211AuthenticatedAssociated
} EFI_80211_LINK_STATE;

Ieee80211UnauthenticatedUnassociated
    Indicates initial start state, unauthenticated, unassociated.

Ieee80211AuthenticatedUnassociated
    Indicates authenticated, unassociated.

Ieee80211PendingRSNAuthentication
    Indicates authenticated and associated, but pending RSN
authentication.

Ieee80211AuthenticatedAssociated
    Indicates authenticated and associated.

,//**********************************************
// EFI_SUPPLICANT_KEY_REFRESH
//**********************************************
typedef struct {
    BOOLEAN       GTKRefresh;
} EFI_SUPPLICANT_KEY_REFRESH;

GTKRefresh
    If TRUE, indicates GTK is just refreshed after a successful call to
**EFI_SUPPLICANT_PROTOCOL.BuildResponsePacket()**.
/****************************************************************************
// EFI_SUPPLICANT_GTK_LIST
//****************************************************************************
typedef struct {
    UINT8 GTKCount;
    EFI_SUPPLICANT_KEY GTKList[1];
} EFI_SUPPLICANT_GTK_LIST;

GTKCount Indicates the number of GTKs that are contained in GTKList.
GTKList A variable-length array of GTKs of type EFI_SUPPLICANT_KEY. The number of entries is specified by GTKCount.

#define EFI_MAX_KEY_LEN 64

 /*******************************************************************************/
// EFI_SUPPLICANT_KEY
/*******************************************************************************/
typedef struct {
    UINT8 Key[EFI_MAX_KEY_LEN];
    UINT8 KeyLen;
    UINT8 KeyId;
    EFI_SUPPLICANT_KEY_TYPE KeyType;
    EFI_80211_MAC_ADDRESS Addr;
    UINT8 Rsc[8];
    UINT8 RscLen;
    BOOLEAN IsAuthenticator;
    EFI_80211_SUITE_SELECTOR CipherSuite;
    EFI_SUPPLICANT_KEY_DIRECTION Direction;
} EFI_SUPPLICANT_KEY;

The EFI_SUPPLICANT_KEY descriptor is defined in the IEEE 802.11 standard, section 6.3.19.1.2.

Key The key value.
KeyLen Length in bytes of the Key. Should be up to EFI_MAX_KEY_LEN.
KeyId The key identifier.
KeyType Defines whether this key is a group key, pairwise key, PeerKey, or Integrity Group.
Addr The value is set according to the KeyType.
RSC The Receive Sequence Count value.
RscLen Length in bytes of the Rsc. Should be up to 8.
IsAuthenticator Indicates whether the key is configured by the Authenticator or Supplicant. The value true indicates Authenticator.
CipherSuite The cipher suite required for this association.
Direction Indicates the direction for which the keys are to be installed.
typedef enum {
    Group,
    Pairwise,
    PeerKey,
    IGTK
} EFI_SUPPLICANT_KEY_TYPE;

The EFI_SUPPLICANT_KEY_TYPE is defined in the IEEE 802.11 specification.

typedef enum {
    Receive,
    Transmit,
    Both
} EFI_SUPPLICANT_KEY_DIRECTION;

Receive Indicates that the keys are being installed for the receive direction.
Transmit Indicates that the keys are being installed for the transmit direction.
Both Indicates that the keys are being installed for both the receive and transmit directions.

Status Codes Returned

<table>
<thead>
<tr>
<th>Status Code</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>EFI_SUCCESS</td>
<td>The Supplicant configuration data is set successfully.</td>
</tr>
<tr>
<td>EFI_INVALID_PARAMETER</td>
<td>One or more of the following conditions is TRUE:</td>
</tr>
<tr>
<td></td>
<td>• Data is NULL.</td>
</tr>
<tr>
<td></td>
<td>• DataSize is 0.</td>
</tr>
<tr>
<td>EFI_UNSUPPORTED</td>
<td>The DataType is unsupported.</td>
</tr>
<tr>
<td>EFI_OUT_OF_RESOURCES</td>
<td>Required system resources could not be allocated.</td>
</tr>
</tbody>
</table>

 EFI_SUPPLICANT_PROTOCOL.GetData()

Summary
Get Supplicant configuration data.
Prototype

typedef
EFI_STATUS
(EIFIAPI *EFI_SUPPLICANT_GET_DATA)(
    IN EFI_SUPPLICANT_PROTOCOL     *This,
    IN EFI_SUPPLICANT_DATA_TYPE   DataType,
    OUT UINT8             *Data, OPTIONAL
    IN OUT UINTN            *DataSize
);

Parameters

  This  Pointer to the EFI_SUPPLICANT_PROTOCOL instance.
  DataType The type of data.
  Data Point to the buffer to hold the data. Ignored if DataSize is 0.
  DataSize Pointer to the buffer size in bytes. On input, it is the buffer size
               provided by the caller. On output, it is the buffer size in fact needed
               to contain the packet.

Description

The GetData() function gets Supplicant configuration. The typical example is PTK and GTK derived from
handshake. The wireless NIC can support software encryption or hardware encryption. If the consumer
uses software encryption, it can call ProcessPacket() to get result. If the consumer supports
hardware encryption, it can get PTK and GTK via GetData() and program to hardware register.

Status Codes Returned

<table>
<thead>
<tr>
<th>Status Code</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>EFI_SUCCESS</td>
<td>The Supplicant configuration data is got successfully.</td>
</tr>
<tr>
<td>EFI_INVALID_PARAMETER</td>
<td>One or more of the following conditions is TRUE:</td>
</tr>
<tr>
<td></td>
<td>• This is NULL.</td>
</tr>
<tr>
<td></td>
<td>• DataSize is NULL.</td>
</tr>
<tr>
<td></td>
<td>• Data is NULL if *DataSize is not zero.</td>
</tr>
<tr>
<td>EFI_UNSUPPORTED</td>
<td>The DataType is unsupported.</td>
</tr>
<tr>
<td>EFI_NOT_FOUND</td>
<td>The Supplicant configuration data is not found.</td>
</tr>
<tr>
<td>EFI_BUFFER_TOO_SMALL</td>
<td>The size of Data is too small for the specified configuration data and the</td>
</tr>
<tr>
<td></td>
<td>required size is returned in DataSize.</td>
</tr>
</tbody>
</table>
28 - Network Protocols — TCP, IP, IPsec, FTP, TLS and Configurations

28.1 EFI TCPv4 Protocol

This section defines the EFI TCPv4 (Transmission Control Protocol version 4) Protocol.

28.1.1 TCP4 Service Binding Protocol

EFI_TCP4_SERVICE_BINDING_PROTOCOL

Summary

The EFI TCPv4 Service Binding Protocol is used to locate EFI TCPv4 Protocol drivers to create and destroy child of the driver to communicate with other host using TCP protocol.

GUID

#define EFI_TCP4_SERVICE_BINDING_PROTOCOL_GUID
  {0x00720665, 0x67EB, 0x4a99, {0xBA, 0xF7, 0xD3, 0xC3, 0x3A, 0x1C, 0x7C, 0xC9}}

Description

A network application that requires TCPv4 I/O services can call one of the protocol handler services, such as BS->LocateHandleBuffer(), to search devices that publish an EFI TCPv4 Service Binding Protocol GUID. Such device supports the EFI TCPv4 Protocol and may be available for use.

After a successful call to the EFI_TCP4_SERVICE_BINDING_PROTOCOL.CreateChild() function, the newly created child EFI TCPv4 Protocol driver is in an un-configured state; it is not ready to do any operation except Poll() send and receive data packets until configured as the purpose of the user and perhaps some other indispensable function belonged to TCPv4 Protocol driver is called properly.

Every successful call to the EFI_TCP4_SERVICE_BINDING_PROTOCOL.CreateChild() function must be matched with a call to the EFI_TCP4_SERVICE_BINDING_PROTOCOL.DestroyChild() function to release the protocol driver.

28.1.2 TCP4 Protocol

EFI_TCP4_PROTOCOL

Summary

The EFI TCPv4 Protocol provides services to send and receive data stream.

GUID

#define EFI_TCP4_PROTOCOL_GUID
  {0x65530BC7, 0xA359, 0x410f, {0xB0, 0x10, 0x5A, 0xAD, 0xC7, 0xEC, 0x2B, 0x62}}
Protocol Interface Structure

typedef struct _EFI_TCP4_PROTOCOL {
    EFI_TCP4_GET_MODE_DATA   GetModeData;
    EFI_TCP4_CONFIGURE       Configure;
    EFI_TCP4_ROUTES          Routes;
    EFI_TCP4_CONNECT         Connect;
    EFI_TCP4_ACCEPT          Accept;
    EFI_TCP4_TRANSMIT        Transmit;
    EFI_TCP4_RECEIVE         Receive;
    EFI_TCP4_CLOSE           Close;
    EFI_TCP4_CANCEL          Cancel;
    EFI_TCP4_POLL            Poll;
} EFI_TCP4_PROTOCOL;

Parameters

GetModeData Get the current operational status. See the GetModeData() function description.

Configure Initialize, change, or brutally reset operational settings of the EFI TCPv4 Protocol. See the Configure() function description.

Routes Add or delete routing entries for this TCP4 instance. See the Routes() function description.

Connect Initiate the TCP three-way handshake to connect to the remote peer configured in this TCP instance. The function is a nonblocking operation. See the Connect() function description.

Accept Listen for incoming TCP connection request. This function is a nonblocking operation. See the Accept() function description.

Transmit Queue outgoing data to the transmit queue. This function is a nonblocking operation. See the Transmit() function description.

Receive Queue a receiving request token to the receive queue. This function is a nonblocking operation. See the Receive() function description.

Close Gracefully disconnecting a TCP connection follow RFC 793 or reset a TCP connection. This function is a nonblocking operation. See the Close() function description.

Cancel Abort a pending connect, listen, transmit or receive request. See the Cancel() function description.

Poll Poll to receive incoming data and transmit outgoing TCP segments. See the Poll() function description.

Description

The EFI_TCP4_PROTOCOL defines the EFI TCPv4 Protocol child to be used by any network drivers or applications to send or receive data stream. It can either listen on a specified port as a service or actively
connected to remote peer as a client. Each instance has its own independent settings, such as the routing table.

**Note:** In this document, all IPv4 addresses and incoming/outgoing packets are stored in network byte order. All other parameters in the functions and data structures that are defined in this document are stored in host byte order unless explicitly specified.

**EFI_TCP4_PROTOCOL.GetModeData()**

**Summary**
Get the current operational status.

**Prototype**
```c
typedef EFI_STATUS
    (EFIAPI *EFI_TCP4_GET_MODE_DATA) (
    IN EFI_TCP4_PROTOCOL *This,
    OUT EFI_TCP4_CONNECTION_STATE *Tcp4State OPTIONAL,
    OUT EFI_TCP4_CONFIG_DATA *Tcp4ConfigData OPTIONAL,
    OUT EFI_IPv4_MODE_DATA *Ip4ModeData OPTIONAL,
    OUT EFI_MANAGED_NETWORK_CONFIG_DATA *MnpConfigData OPTIONAL,
    OUT EFI_SIMPLE_NETWORK_MODE *SnpModeData OPTIONAL);
```

**Parameters**
- **This** Pointer to the EFI_TCP4_PROTOCOL instance.
- **Tcp4State** Pointer to the buffer to receive the current TCP state. Type EFI_TCP4_CONNECTION_STATE is defined in “Related Definitions” below.
- **Tcp4ConfigData** Pointer to the buffer to receive the current TCP configuration. Type EFI_TCP4_CONFIG_DATA is defined in “Related Definitions” below.
- **Ip4ModeData** Pointer to the buffer to receive the current IPv4 configuration data used by the TCPv4 instance. Type EFI_IP4_MODE_DATA is defined in EFI_IP4_PROTOCOL.GetModeData().
- **MnpConfigData** Pointer to the buffer to receive the current MNP configuration data used indirectly by the TCPv4 instance. Type EFI_MANAGED_NETWORK_CONFIG_DATA is defined in EFI_MANAGED_NETWORK_PROTOCOL.GetModeData().
- **SnpModeData** Pointer to the buffer to receive the current SNP configuration data used indirectly by the TCPv4 instance. Type EFI_SIMPLE_NETWORK_MODE is defined in the EFI_SIMPLE_NETWORK_PROTOCOL.
Description
The `GetModeData()` function copies the current operational settings of this EFI TCPv4 Protocol instance into user-supplied buffers. This function can also be used to retrieve the operational setting of underlying drivers such as IPv4, MNP, or SNP.

Related Definition
```c
typedef struct {
    BOOLEAN UseDefaultAddress;
    EFI_IPv4_ADDRESS StationAddress;
    EFI_IPv4_ADDRESS SubnetMask;
    UINT16 StationPort;
    EFI_IPv4_ADDRESS RemoteAddress;
    UINT16 RemotePort;
    BOOLEAN ActiveFlag;
} EFI_TCP4_ACCESS_POINT;
```

- **UseDefaultAddress**
  Set to `TRUE` to use the default IP address and default routing table. If the default IP address is not available yet, then the underlying EFI IPv4 Protocol driver will use `EFI_IP4_CONFIG2_PROTOCOL` to retrieve the IP address and subnet information.

- **StationAddress**
  The local IP address assigned to this EFI TCPv4 Protocol instance. The EFI TCPv4 and EFI IPv4 Protocol drivers will only deliver incoming packets whose destination addresses exactly match the IP address. Not used when `UseDefaultAddress` is `TRUE`.

- **SubnetMask**
  The subnet mask associated with the station address. Not used when `UseDefaultAddress` is `TRUE`.

- **StationPort**
  The local port number to which this EFI TCPv4 Protocol instance is bound. If the instance doesn't care the local port number, set `StationPort` to zero to use an ephemeral port.

- **RemoteAddress**
  The remote IP address to which this EFI TCPv4 Protocol instance is connected. If `ActiveFlag` is `FALSE` (i.e., a passive TCPv4 instance), the instance only accepts connections from the `RemoteAddress`. If `ActiveFlag` is `TRUE` the instance is connected to the `RemoteAddress`, i.e., outgoing segments will be sent to this address and only segments from this address will be delivered to the application. When `ActiveFlag` is `FALSE` it can be set to zero and means that incoming connection request from any address will be accepted.

- **RemotePort**
  The remote port to which this EFI TCPv4 Protocol instance is connects or connection request from which is accepted by this EFI TCPv4 Protocol instance. If `ActiveFlag` is `FALSE` it can be zero and means that incoming connection request from any port will be accepted. Its value can not be zero when `ActiveFlag` is `TRUE`.

- **ActiveFlag**
  Set it to `TRUE` to initiate an active open. Set it to `FALSE` to initiate a passive open to act as a server.
typedef struct {
  UINT32 ReceiveBufferSize;
  UINT32 SendBufferSize;
  UINT32 MaxSynBackLog;
  UINT32 ConnectionTimeout;
  UINT32 DataRetries;
  UINT32 FinTimeout;
  UINT32 TimeWaitTimeout;
  UINT32 KeepAliveProbes;
  UINT32 KeepAliveTime;
  UINT32 KeepAliveInterval;
  BOOLEAN EnableNagle;
  BOOLEAN EnableTimeStamp;
  BOOLEAN EnableWindowScaling;
  BOOLEAN EnableSelectiveAck;
  BOOLEAN EnablePathMtuDiscovery;
} EFI_TCP4_OPTION;

ReceiveBufferSize The size of the TCP receive buffer.
SendBufferSize The size of the TCP send buffer.
MaxSynBackLog The length of incoming connect request queue for a passive instance. When set to zero, the value is implementation specific.
ConnectionTimeout The maximum seconds a TCP instance will wait for before a TCP connection established. When set to zero, the value is implementation specific.
DataRetries The number of times TCP will attempt to retransmit a packet on an established connection. When set to zero, the value is implementation specific.
FinTimeout How many seconds to wait in the FIN_WAIT_2 states for a final FIN flag before the TCP instance is closed. This timeout is in effective only if the application has called Close() to disconnect the connection completely. It is also called FIN_WAIT_2 timer in other implementations. When set to zero, it should be disabled because the FIN_WAIT_2 timer itself is against the standard.
TimeWaitTimeout How many seconds to wait in TIME_WAIT state before the TCP instance is closed. The timer is disabled completely to provide a method to close the TCP connection quickly if it is set to zero. It is against the related RFC documents.
KeepAliveProbes The maximum number of TCP keep-alive probes to send before giving up and resetting the connection if no response from the other end. Set to zero to disable keep-alive probe.
KeepAliveTime The number of seconds a connection needs to be idle before TCP sends out periodical keep-alive probes. When set to zero, the value is implementation specific. It should be ignored if keep-alive probe is disabled.

KeepAliveInterval The number of seconds between TCP keep-alive probes after the periodical keep-alive probe if no response. When set to zero, the value is implementation specific. It should be ignored if keep-alive probe is disabled.

EnableNagle Set it to **TRUE** to enable the Nagle algorithm as defined in RFC896. Set it to **FALSE** to disable it.

EnableTimeStamp Set it to **TRUE** to enable TCP timestamps option as defined in RFC7323. Set to **FALSE** to disable it.

EnableWindowScaling Set it to **TRUE** to enable TCP window scale option as defined in RFC7323. Set to **FALSE** to disable it.

EnableSelectiveAck Set it to **TRUE** to enable selective acknowledge mechanism described in RFC 2018. Set it to **FALSE** to disable it. Implementation that supports SACK can optionally support DSAK as defined in RFC 2883.

EnablePathMtudiscovery Set it to **TRUE** to enable path MTU discovery as defined in RFC 1191. Set to **FALSE** to disable it.

Option setting with digital value will be modified by driver if it is set out of the implementation specific range and an implementation specific default value will be set accordingly.

```c
//**************************************************************
// EFI_TCP4_CONFIG_DATA
//**************************************************************
typedef struct {
    // Receiving Filters
    // I/O parameters
    UINT8 TypeOfService;
    UINT8 TimeToLive;

    // Access Point
    EFI_TCP4_ACCESS_POINT AccessPoint;

    // TCP Control Options
    EFI_TCP4_OPTION * ControlOption;
} EFI_TCP4_CONFIG_DATA;
```

**TypeOfService** field in transmitted IPv4 packets.

**TimeToLive** field in transmitted IPv4 packets.
AccessPoint  Used to specify TCP communication end settings for a TCP instance.

ControlOption  Used to configure the advance TCP option for a connection. If set to NULL, implementation specific options for TCP connection will be used.

//***************************************************************
// EFI_TCP4_CONNECTION_STATE
//***************************************************************/

typedef enum {
    Tcp4StateClosed    = 0,
    Tcp4StateListen    = 1,
    Tcp4StateSynSent    = 2,
    Tcp4StateSynReceived  = 3,
    Tcp4StateEstablished  = 4,
    Tcp4StateFinWait1   = 5,
    Tcp4StateFinWait2   = 6,
    Tcp4StateClosing    = 7,
    Tcp4StateTimeWait   = 8,
    Tcp4StateCloseWait   = 9,
    Tcp4StateLastAck    = 10
} EFI_TCP4_CONNECTION_STATE;

Status Codes Returned

<table>
<thead>
<tr>
<th>Status Code</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>EFI_SUCCESS</td>
<td>The mode data was read.</td>
</tr>
<tr>
<td>EFI_NOT_STARTED</td>
<td>No configuration data is available because this instance hasn’t been started.</td>
</tr>
<tr>
<td>EFI_INVALID_PARAMETER</td>
<td>This is NULL.</td>
</tr>
</tbody>
</table>

EFI_TCP4_PROTOCOL.Configure()

Summary

Initialize or brutally reset the operational parameters for this EFI TCPv4 instance.

Prototype

typedef
EFI_STATUS
(EIFIAPI *EFI_TCP4_CONFIGURE) (  
    IN EFI_TCP4_PROTOCOL  *This,
    IN EFI_TCP4_CONFIG_DATA  *TcpConfigData OPTIONAL
);

Parameters

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>This</td>
<td>Pointer to the EFI_TCP4_PROTOCOL instance.</td>
</tr>
<tr>
<td>TcpConfigData</td>
<td>Pointer to the configure data to configure the instance.</td>
</tr>
</tbody>
</table>
Description

The `Configure()` function does the following:

- Initialize this EFI TCPv4 instance, i.e., initialize the communication end setting, specify active open or passive open for an instance.
- Reset this TCPv4 instance brutally, i.e., cancel all pending asynchronous tokens, flush transmission and receiving buffer directly without informing the communication peer.

No other TCPv4 Protocol operation can be executed by this instance until it is configured properly. For an active TCP4 instance, after a proper configuration it may call `Connect()` to initiates the three-way handshake. For a passive TCP4 instance, its state will transit to `Tcp4StateListen` after configuration, and `Accept()` may be called to listen the incoming TCP connection request. If `TcpConfigData` is set to NULL, the instance is reset. Resetting process will be done brutally, the state machine will be set to `Tcp4StateClosed` directly, the receive queue and transmit queue will be flushed, and no traffic is allowed through this instance.

Status Codes Returned

<table>
<thead>
<tr>
<th>Status Code</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>EFI_SUCCESS</td>
<td>The operational settings are set, changed, or reset successfully.</td>
</tr>
<tr>
<td>EFI_NO_MAPPING</td>
<td>When using a default address, configuration (through DHCP, BOOTP, RARP, etc.) is not finished yet.</td>
</tr>
</tbody>
</table>
| EFI_INVALID_PARAMETER        | One or more following conditions are **TRUE**:
- **This** is **NULL**
- `TcpConfigData` `->AccessPoint.StationAddress` isn’t a valid unicast IPv4 address when `TcpConfigData` `->AccessPoint.UseDefaultAddress` is **FALSE**.
- `TcpConfigData` `->AccessPoint.SubnetMask` isn’t a valid IPv4 address mask when `TcpConfigData` `-> AccessPoint.UseDefaultAddress` is **FALSE**. The subnet mask must be contiguous.
- `TcpConfigData` `->AccessPoint.RemoteAddress` isn’t a valid unicast IPv4 address.
- `TcpConfigData` `->AccessPoint.RemoteAddress` is zero or `TcpConfigData` `->AccessPoint.RemotePort` is zero when `TcpConfigData` `->AccessPoint.ActiveFlag` is **TRUE**.
- A same access point has been configured in other TCP instance properly. |
| EFI_ACCESS_DENIED            | Configuring TCP instance when it is configured without calling `Configure()` with NULL to reset it. |
| EFI_DEVICE_ERROR             | An unexpected network or system error occurred. |
EFI_TCP4_PROTOCOL.Routes()

Summary
Add or delete routing entries.

Prototype
```c
typedef EFI_STATUS
    (EFIAPI *EFI_TCP4_ROUTES) (  
        IN EFI_TCP4_PROTOCOL *This,
        IN BOOLEAN DeleteRoute,
        IN EFI_IPv4_ADDRESS *SubnetAddress,
        IN EFI_IPv4_ADDRESS *SubnetMask,
        IN EFI_IPv4_ADDRESS *GatewayAddress
    );
```

Parameters
- This
  Pointer to the EFI_TCP4_PROTOCOL instance.
- DeleteRoute
  Set it to TRUE to delete this route from the routing table. Set it to FALSE to add this route to the routing table. 
  DestinationAddress and SubnetMask are used as the keywords to search route entry.
- SubnetAddress
  The destination network.
- SubnetMask
  The subnet mask of the destination network.
- GatewayAddress
  The gateway address for this route. It must be on the same subnetwork with the station address unless a direct route is specified.

Description
The Routes() function adds or deletes a route from the instance’s routing table.

The most specific route is selected by comparing the SubnetAddress with the destination IP address’s arithmetical AND to the SubnetMask.

The default route is added with both SubnetAddress and SubnetMask set to 0.0.0.0. The default route matches all destination IP addresses if there is no more specific route.

Direct route is added with GatewayAddress set to 0.0.0.0. Packets are sent to the destination host if its address can be found in the Address Resolution Protocol (ARP) cache or it is on the local subnet. If the instance is configured to use default address, a direct route to the local network will be added automatically.
Each TCP instance has its own independent routing table. Instance that uses the default IP address will have a copy of the EFI_IP4_CONFIG2_PROTOCOL’s routing table. The copy will be updated automatically whenever the IP driver reconfigures its instance. As a result, the previous modification to the instance’s local copy will be lost.

The priority of checking the route table is specific with IP implementation and every IP implementation must comply with RFC 1122.

**Note:** There is no way to set up routes to other network interface cards (NICs) because each NIC has its own independent network stack that shares information only through EFI TCP4 variable.

### Status Codes Returned

<table>
<thead>
<tr>
<th>Code</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>EFI_SUCCESS</td>
<td>The operation completed successfully.</td>
</tr>
<tr>
<td>EFI_NOT_STARTED</td>
<td>The EFI TCPv4 Protocol instance has not been configured.</td>
</tr>
<tr>
<td>EFI_NO_MAPPING</td>
<td>When using a default address, configuration (DHCP, BOOTP, RARP, etc.) is not finished yet.</td>
</tr>
<tr>
<td>EFI_INVALID_PARAMETER</td>
<td>One or more of the following conditions is TRUE:</td>
</tr>
<tr>
<td></td>
<td>• This is <strong>NULL</strong>.</td>
</tr>
<tr>
<td></td>
<td>• SubnetAddress is <strong>NULL</strong>.</td>
</tr>
<tr>
<td></td>
<td>• SubnetMask is <strong>NULL</strong>.</td>
</tr>
<tr>
<td></td>
<td>• GatewayAddress is <strong>NULL</strong>.</td>
</tr>
<tr>
<td></td>
<td>• *SubnetAddress is not <strong>NULL</strong> a valid subnet address.</td>
</tr>
<tr>
<td></td>
<td>• *SubnetMask is not a valid subnet mask.</td>
</tr>
<tr>
<td></td>
<td>• *GatewayAddress is not a valid unicast IP address or it is not in the same subnet.</td>
</tr>
<tr>
<td>EFI_OUT_OF_RESOURCES</td>
<td>Could not allocate enough resources to add the entry to the routing table.</td>
</tr>
<tr>
<td>EFI_NOT_FOUND</td>
<td>This route is not in the routing table.</td>
</tr>
<tr>
<td>EFI_ACCESS_DENIED</td>
<td>The route is already defined in the routing table.</td>
</tr>
<tr>
<td>EFI_UNSUPPORTED</td>
<td>The TCP driver does not support this operation.</td>
</tr>
</tbody>
</table>

### EFI_TCP4_PROTOCOL.Connect()

#### Summary

Initiate a nonblocking TCP connection request for an active TCP instance.

#### Prototype

```c
typedef EFI_STATUS
(EFIAPI *EFI_TCP4_CONNECT) (    
    IN EFI_TCP4_PROTOCOL    *This,
    IN EFI_TCP4_CONNECTION_TOKEN  *ConnectionToken,
    ...);
```
Parameters

This
    Pointer to the EFI_TCP4_PROTOCOL instance.
ConnectionToken
    Pointer to the connection token to return when the TCP three way handshake finishes. Type EFI_TCP4_CONNECTION_TOKEN is defined in “Related Definition” below.

Description

The Connect() function will initiate an active open to the remote peer configured in current TCP instance if it is configured active. If the connection succeeds or fails due to any error, the ConnectionToken->CompletionToken.Event will be signaled and ConnectionToken->CompletionToken.Status will be updated accordingly. This function can only be called for the TCP instance in Tcp4StateClosed state. The instance will transfer into Tcp4StateSynSent if the function returns EFI_SUCCESS. If TCP three way handshake succeeds, its state will become Tcp4StateEstablished, otherwise, the state will return to Tcp4StateClosed.

Related Definitions

typedef struct {
    EFI_EVENT Event;
    EFI_STATUS Status;
} EFI_TCP4_COMPLETION_TOKEN;

Event
    The Event to signal after request is finished and Status field is updated by the EFI TCPv4 Protocol driver. The type of Event must be EVT_NOTIFY_SIGNAL, and its Task Priority Level (TPL) must be lower than or equal to TPL_CALLBACK.

Status
    The variable to receive the result of the completed operation. EFI_NO_MEDIA. There was a media error

The EFI_TCP4_CONNECTION_TOKEN is used as a common header for various asynchronous tokens.

typedef struct {
    EFI_TCP4_COMPLETION_TOKEN CompletionToken;
} EFI_TCP4_CONNECTION_TOKEN;

Status
    The Status in the CompletionToken will be set to one of the following values if the active open succeeds or an unexpected error happens:
**EFI_SUCCESS.** The active open succeeds and the instance is in `Tcp4StateEstablished`.

**EFI_CONNECTION_RESET.** The connect fails because the connection is reset either by instance itself or communication peer.

**EFI_CONNECTION_REFUSED.** The connect fails because this connection is initiated with an active open and the connection is refused.

**EFI_ABORTED.** The active open was aborted.

**EFI_TIMEOUT.** The connection establishment timer expired and no more specific information is available.

**EFI_NETWORK_UNREACHABLE.** The active open fails because an ICMP network unreachable error is received.

**EFI_HOST_UNREACHABLE.** The active open fails because an ICMP host unreachable error is received.

**EFI_PROTOCOL_UNREACHABLE.** The active open fails because an ICMP protocol unreachable error is received.

**EFI_PORT_UNREACHABLE.** The connection establishment timer times out and an ICMP port unreachable error is received.

**EFI_ICMP_ERROR.** The connection establishment timer timeout and some other ICMP error is received.

**EFI_DEVICE_ERROR.** An unexpected system or network error occurred.

### Status Codes Returned

<table>
<thead>
<tr>
<th>Code</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>EFI_SUCCESS</td>
<td>The connection request is successfully initiated and the state of this TCPv4 instance has been changed to <code>Tcp4StateSynSent</code></td>
</tr>
<tr>
<td>EFI_NOT_STARTED</td>
<td>This EFI TCPv4 Protocol instance has not been configured.</td>
</tr>
</tbody>
</table>
| EFI_ACCESS_DENIED   | One or more of the following conditions are **TRUE**:  
  - This instance is not configured as an active one.  
  - This instance is not in `Tcp4StateClosed` state. |
| EFI_INVALID_PARAMETER | One or more of the following are **TRUE**:  
  - This is **NULL**  
  - `ConnectionToken` is **NULL**  
  - `ConnectionToken` ->`CompletionToken.Event` is **NULL**. |
| EFI_OUT_OF_RESOURCES | The driver can’t allocate enough resource to initiate the active open. |
| EFI_DEVICE_ERROR    | An unexpected system or network error occurred. |
**EFI_TCP4_PROTOCOL.Accept()**

**Summary**
Listen on the passive instance to accept an incoming connection request. This is a nonblocking operation.

**Prototype**
```
typedef
EFI_STATUS
(EIFIAPI *EFI_TCP4_ACCEPT) (  
    IN EFI_TCP4_PROTOCOL *This,
    IN EFI_TCP4_LISTEN_TOKEN *ListenToken
);
```

**Parameters**
- **This**: Pointer to the EFI_TCP4_PROTOCOL instance.
- **ListenToken**: Pointer to the listen token to return when operation finishes. Type EFI_TCP4_LISTEN_TOKEN is defined in “Related Definition” below.

**Related Definitions**
```c
//******************************************************************************
// EFI_TCP4_LISTEN_TOKEN
//******************************************************************************
typedef struct {
    EFI_TCP4_COMPLETION_TOKEN CompletionToken;
    EFI_HANDLE NewChildHandle;
} EFI_TCP4_LISTEN_TOKEN;
```

**Status**
The Status in CompletionToken will be set to the following value if accept finishes:

- **EFI_SUCCESS**: A remote peer has successfully established a connection to this instance. A new TCP instance has also been created for the connection.
- **EFI_CONNECTION_RESET**: The accept fails because the connection is reset either by instance itself or communication peer.
- **EFI_ABORTED**: The accept request has been aborted.

**NewChildHandle**
The new TCP instance handle created for the established connection.

**Description**
The **Accept()** function initiates an asynchronous accept request to wait for an incoming connection on the passive TCP instance. If a remote peer successfully establishes a connection with this instance, a new TCP instance will be created and its handle will be returned in ListenToken->NewChildHandle. The newly created
instance is configured by inheriting the passive instance’s configuration and is ready for use upon return. The instance is in the **Tcp4StateEstablished** state.

The `ListenToken->CompletionToken.Event` will be signaled when a new connection is accepted, user aborts the listen or connection is reset.

This function only can be called when current TCP instance is in **Tcp4StateListen** state.

**Status Codes Returned**

<table>
<thead>
<tr>
<th>Code</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>EFI_SUCCESS</td>
<td>The listen token has been queued successfully.</td>
</tr>
<tr>
<td>EFI_NOT_STARTED</td>
<td>This EFI TCPv4 Protocol instance has not been configured.</td>
</tr>
</tbody>
</table>
| EFI_ACCESS_DENIED | One or more of the following are **TRUE**:  
  - This instance is not a passive instance.  
  - This instance is not in **Tcp4StateListen** state.  
  - The same listen token has already existed in the listen token queue of this TCP instance. |
| EFI_INVALID_PARAMETER | One or more of the following are **TRUE**:  
  - **This** is **NULL**.  
  - **ListenToken** is **NULL**.  
  - **ListenToken->CompletionToken.Event** is **NULL**. |
| EFI_OUT_OF_RESOURCES | Could not allocate enough resource to finish the operation. |
| EFI_DEVICE_ERROR | Any unexpected and not belonged to above category error. |

**EFI_TCP4_PROTOCOL.Transmit()**

**Summary**

Queues outgoing data into the transmit queue.

**Prototype**

```c
typedef EFI_STATUS (EFI_API *EFI_TCP4_TRANSMIT) (  
  IN EFI_TCP4_PROTOCOL *This,  
  IN EFI_TCP4_IO_TOKEN *Token
);
```

**Parameters**

- **This**  
  Pointer to the **EFI_TCP4_PROTOCOL** instance.
- **Token**  
  Pointer to the completion token to queue to the transmit queue. Type **EFI_TCP4_IO_TOKEN** is defined in “Related Definitions” below.
Description

The **Transmit()** function queues a sending request to this TCPv4 instance along with the user data. The status of the token is updated and the event in the token will be signaled once the data is sent out or some error occurs.

Related Definitions

```c
//***************************************************************
// EFI_TCP4_IO_TOKEN
//----------------------------------------------------------------------------
typedef struct {
    EFI_TCP4_COMPLETION_TOKEN    CompletionToken;
    union {
        EFI_TCP4_RECEIVE_DATA    *RxData;
        EFI_TCP4_TRANSMIT_DATA   *TxData;
    }
} EFI_TCP4_IO_TOKEN;
```

**Status**

When transmission finishes or meets any unexpected error it will be set to one of the following values:

- ** EFI_SUCCESS:** The receiving or transmission operation completes successfully.
- ** EFI_CONNECTION_FIN:** The receiving operation fails because the communication peer has closed the connection and there is no more data in the receive buffer of the instance.
- ** EFI_CONNECTION_RESET:** The receiving or transmission operation fails because this connection is reset either by instance itself or communication peer.
- ** EFI_ABORTED:** The receiving or transmission is aborted.
- ** EFI_TIMEOUT:** The transmission timer expires and no more specific information is available.
- ** EFI_NETWORK_UNREACHABLE:** The transmission fails because an ICMP network unreachable error is received.
- ** EFI_HOST_UNREACHABLE:** The transmission fails because an ICMP host unreachable error is received.
- ** EFI_PROTOCOL_UNREACHABLE:** The transmission fails because an ICMP protocol unreachable error is received.
- ** EFI_PORT_UNREACHABLE:** The transmission fails and an ICMP port unreachable error is received.
- ** EFI_ICMP_ERROR:** The transmission fails and some other ICMP error is received.
- ** EFI_DEVICE_ERROR:** An unexpected system or network error occurs.
- ** EFI_NO_MEDIA:** There was a media error.
RxData
When this token is used for receiving, RxData is a pointer to EFI_TCP4_RECEIVE_DATA. Type EFI_TCP4_RECEIVE_DATA is defined below.

TxData
When this token is used for transmitting, TxData is a pointer to EFI_TCP4_TRANSMIT_DATA. Type EFI_TCP4_TRANSMIT_DATA is defined below.

The EFI_TCP4_IO_TOKEN structures are used for both transmit and receive operations.

When used for transmitting, the CompletionToken.Event and TxData fields must be filled in by the user. After the transmit operation completes, the CompletionToken.Status field is updated by the instance and the Event is signaled.

• When used for receiving, the CompletionToken.Event and RxData fields must be filled in by the user. After a receive operation completes, RxData and Status are updated by the instance and the Event is signaled.

***************************************************************
// TCP4 Token Status definition
//***************************************************************
#define EFI_CONNECTION_FIN        EFIERR (104)
#define EFI_CONNECTION_RESET       EFIERR (105)
#define EFI_CONNECTION_REFUSED      EFIERR (106)

Note: EFIERR() sets the maximum bit. Similar to how error codes are described in Appendix D.

//******************************************************************************
// EFI_TCP4_RECEIVE_DATA
//******************************************************************************
typedef struct {
   BOOLEAN UrgentFlag;
   UINT32  DataLength;
   UINT32  FragmentCount;
   EFI_TCP4_FRAGMENT_DATA FragmentTable[1];
} EFI_TCP4_RECEIVE_DATA;

UrgentFlag
Whether those data are urgent. When this flag is set, the instance is in urgent mode. The implementations of this specification should follow RFC793 to process urgent data, and should NOT mix the data across the urgent point in one token.

DataLength
When calling Receive() function, it is the byte counts of all Fragmentbuffer in FragmentTable allocated by user. When the token is signaled by TCPv4 driver it is the length of received data in the fragments.
FragmentCount  Number of fragments.
FragmentTable  An array of fragment descriptors. Type EFI_TCP4_FRAGMENT_DATA is defined below.

When TCPv4 driver wants to deliver received data to the application, it will pick up the first queued receiving token, update its Token->Packet.RxData then signal the Token->CompletionToken.Event.

- The FragmentBuffers in FragmentTable are allocated by the application when calling Receive() function and received data will be copied to those buffers by the driver. FragmentTable may contain multiple buffers that are NOT in the continuous memory locations. The application should combine those buffers in the FragmentTable to process data if necessary.

```c
//***************************************************************
// EFI_TCP4_FRAGMENT_DATA
//***************************************************************
typedef struct {
    UINT32    FragmentLength;
    VOID*     *FragmentBuffer;
} EFI_TCP4_FRAGMENT_DATA;
```

FragmentLength  Length of data buffer in the fragment.
FragmentBuffer  Pointer to the data buffer in the fragment.

EFI_TCP4_FRAGMENT_DATA allows multiple receive or transmit buffers to be specified. The purpose of this structure is to provide scattered read and write.

```c
//***************************************************************
// EFI_TCP4_TRANSMIT_DATA
//***************************************************************
typedef struct {
    BOOLEAN    Push;
    BOOLEAN    Urgent;
    UINT32     DataLength;
    UINT32     FragmentCount;
    EFI_TCP4_FRAGMENT_DATA FragmentTable[1];
} EFI_TCP4_TRANSMIT_DATA;
```

Push  If TRUE, data must be transmitted promptly, and the PUSH bit in the last TCP segment created will be set. If FALSE, data transmission may be delay to combine with data from subsequent Transmit()s for efficiency.
Urgent

The data in the fragment table are urgent and urgent point is in effect if TRUE. Otherwise those data are NOT considered urgent.

DataLength
Length of the data in the fragments.

FragmentCount
Number of fragments.

FragmentTable
A array of fragment descriptors. Type EFI_TCP4_FRAGMENT_DATA is defined above.

The EFI TCPv4 Protocol user must fill this data structure before sending a packet. The packet may contain multiple buffers in non-continuous memory locations.

Status Codes Returned

<table>
<thead>
<tr>
<th>Code</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>EFI_SUCCESS</td>
<td>The data has been queued for transmission.</td>
</tr>
<tr>
<td>EFI_NOT_STARTED</td>
<td>This EFI TCPv4 Protocol instance has not been configured.</td>
</tr>
<tr>
<td>EFI_NO_MAPPING</td>
<td>When using a default address, configuration (DHCP, BOOTP, RARP, etc.) is not finished yet.</td>
</tr>
<tr>
<td>EFI_INVALID_PARAMETER</td>
<td>One or more of the following are TRUE:</td>
</tr>
<tr>
<td></td>
<td>• This is NULL.</td>
</tr>
<tr>
<td></td>
<td>• Token is NULL.</td>
</tr>
<tr>
<td></td>
<td>• Token-&gt;CompletionToken.Event is NULL.</td>
</tr>
<tr>
<td></td>
<td>• Token-&gt;Packet.TxData is NULL.</td>
</tr>
<tr>
<td></td>
<td>• Token-&gt;Packet.FragmentCount is zero.</td>
</tr>
<tr>
<td></td>
<td>• Token-&gt;Packet.DataLength is not equal to the sum of fragment lengths.</td>
</tr>
<tr>
<td>EFI_ACCESS_DENIED</td>
<td>One or more of the following conditions is TRUE:</td>
</tr>
<tr>
<td></td>
<td>• A transmit completion token with the same Token-&gt;CompletionToken.Event was already in the transmission queue.</td>
</tr>
<tr>
<td></td>
<td>• The current instance is in Tcp4StateClosed state.</td>
</tr>
<tr>
<td></td>
<td>• The current instance is a passive one and it is in Tcp4StateListen state.</td>
</tr>
<tr>
<td></td>
<td>• User has called Close() to disconnect this connection.</td>
</tr>
<tr>
<td>EFI_NOT_READY</td>
<td>The completion token could not be queued because the transmit queue is full.</td>
</tr>
<tr>
<td>EFI_OUT_OF_RESOURCES</td>
<td>Could not queue the transmit data because of resource shortage.</td>
</tr>
<tr>
<td>EFI_NETWORK_UNREACHABLE</td>
<td>There is no route to the destination network or address.</td>
</tr>
<tr>
<td>EFI_NO_MEDIA</td>
<td>There was a media error.</td>
</tr>
</tbody>
</table>

EFI_TCP4_PROTOCOL.Receive()

Summary
Places an asynchronous receive request into the receiving queue.
Prototype

```
typedef
 EFI_STATUS
 (EFIAPI *EFI_TCP4_RECEIVE) (
   IN EFI_TCP4_PROTOCOL *This,
   IN EFI_TCP4_IO_TOKEN *Token
 );
```

Parameters

This Pointer to the EFI_TCP4_PROTOCOL instance.
Token Pointer to a token that is associated with the receive data descriptor. Type EFI_TCP4_IO_TOKEN is defined in EFI_TCP4_PROTOCOL.Transmit().

Description

The Receive() function places a completion token into the receive packet queue. This function is always asynchronous. The caller must allocate the Token->CompletionToken.Event and the FragmentBuffer used to receive data. He also must fill the DataLength which represents the whole length of all FragmentBuffer. When the receive operation completes, the EFI TCPv4 Protocol driver updates the Token->CompletionToken.Status and Token->Packet.RxData fields and the Token->CompletionToken.Event is signaled. If got data the data and its length will be copy into the FragmentTable, in the same time the full length of received data will be recorded in the DataLength fields. Providing a proper notification function and context for the event will enable the user to receive the notification and receiving status. That notification function is guaranteed to not be re-entered.

Status Codes Returned

<table>
<thead>
<tr>
<th>Code</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>EFI_SUCCESS</td>
<td>The receive completion token was cached.</td>
</tr>
<tr>
<td>EFI_NOT_STARTED</td>
<td>This EFI TCPv4 Protocol instance has not been configured.</td>
</tr>
<tr>
<td>EFI_NO_MAPPING</td>
<td>When using a default address, configuration (DHCP, BOOTP, RARP, etc.) is not finished yet.</td>
</tr>
<tr>
<td>EFI_INVALID_PARAMETER</td>
<td>One or more of the following conditions is TRUE:</td>
</tr>
<tr>
<td></td>
<td>• This is NULL.</td>
</tr>
<tr>
<td></td>
<td>• Token is NULL.</td>
</tr>
<tr>
<td></td>
<td>• Token-&gt;CompletionToken.Event is NULL.</td>
</tr>
<tr>
<td></td>
<td>• Token-&gt;Packet.RxData is NULL.</td>
</tr>
<tr>
<td></td>
<td>• Token-&gt;Packet.RxData-&gt;DataLength is 0.</td>
</tr>
<tr>
<td></td>
<td>• The Token-&gt;Packet.RxData-&gt;DataLength is not the sum of all FragmentBuffer length in FragmentTable.</td>
</tr>
<tr>
<td>EFI_OUT_OF_RESOURCES</td>
<td>The receive completion token could not be queued due to a lack of system resources (usually memory).</td>
</tr>
<tr>
<td>EFI_DEVICE_ERROR</td>
<td>An unexpected system or network error occurred. The EFI TCPv4 Protocol instance has been reset to startup defaults.</td>
</tr>
</tbody>
</table>
**EFI_TCP4_PROTOCOL.Close()**

**Summary**
Disconnecting a TCP connection gracefully or reset a TCP connection. This function is a nonblocking operation.

**Prototype**

```c
typedef
EFI_STATUS
(EIFIAPI *EFI_TCP4_CLOSE)(
    IN EFI_TCP4_PROTOCOL *This,
    IN EFI_TCP4_CLOSE_TOKEN *CloseToken
);
```

**Parameters**

- **This**
  Pointer to the `EFI_TCP4_PROTOCOL` instance.
- **CloseToken**
  Pointer to the close token to return when operation finishes.

**Related Definitions**

```c
//******************************************************************************
// EFI_TCP4_CLOSE_TOKEN
//******************************************************************************
typedef struct {
    EFI_TCP4_COMPLETION_TOKEN CompletionToken;
    BOOLEAN AbortOnClose;
} EFI_TCP4_CLOSE_TOKEN;
```

**Status**
When close finishes or meets any unexpected error it will be set to one of the following values:

- **EFI_SUCCESS.** The close operation completes successfully.
- **EFI_ACCESS_DENIED** One or more of the following conditions is `TRUE`:
  - A receive completion token with the same `Token- >CompletionToken.Event` was already in the receive queue.
  - The current instance is in `Tcp4StateClosed` state.
  - The current instance is a passive one and it is in `Tcp4StateListen` state.
  - User has called `Close()` to disconnect this connection.
- **EFI_CONNECTION_FIN** The communication peer has closed the connection and there is no any buffered data in the receive buffer of this instance.
- **EFI_NOT_READY** The receive request could not be queued because the receive queue is full.
- **EFI_NO_MEDIA** There was a media error.
**AbortOnClose**

Abort the TCP connection on close instead of the standard TCP close process when it is set to **TRUE**. This option can be used to satisfy a fast disconnect.

**Description**

Initiate an asynchronous close token to TCP driver. After **Close()** is called, any buffered transmission data will be sent by TCP driver and the current instance will have a graceful close working flow described as RFC 793 if **AbortOnClose** is set to **FALSE**, otherwise, a rest packet will be sent by TCP driver to fast disconnect this connection. When the close operation completes successfully the TCP instance is in **Tcp4StateClosed** state, all pending asynchronous operation is signaled and any buffers used for TCP network traffic is flushed.

**Status Codes Returned**

<table>
<thead>
<tr>
<th>Status Code</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>EFI_SUCCESS</td>
<td>The <strong>Close()</strong> is called successfully.</td>
</tr>
<tr>
<td>EFI_NOT_STARTED</td>
<td>This EFI TCPv4 Protocol instance has not been configured.</td>
</tr>
</tbody>
</table>
| EFI_ACCESS_DENIED | One or more of the following are **TRUE**:  
  - **Configure()** has been called with **TcpConfigData** set to NULL and this function has not returned.  
  - Previous **Close()** call on this instance has not finished. |
| EFI_INVALID_PARAMETER | One or more of the following are **TRUE**:  
  - **This** is NULL  
  - **CloseToken** is NULL  
  - **CloseToken->CompletionToken.Event** is NULL |
| EFI_OUT_OF_RESOURCES | Could not allocate enough resource to finish the operation. |
| EFI_DEVICE_ERROR | Any unexpected and not belonged to above category error. |

**EFI_TCP4_PROTOCOL.Cancel()**

**Summary**

Abort an asynchronous connection, listen, transmission or receive request.

**Prototype**

```c
typedef EFI_STATUS
  (EFIAPI *EFI_TCP4_CANCEL)(
    IN EFI_TCP4_PROTOCOL *This,
    IN EFI_TCP4_COMPLETION_TOKEN *Token OPTIONAL);
```

**Parameters**

- **This**  
  Pointer to the **EFI_TCP4_PROTOCOL** instance.
Token

Pointer to a token that has been issued by
EFI_TCP4_PROTOCOL.Connect(),
EFI_TCP4_PROTOCOL.Accept(),
EFI_TCP4_PROTOCOL.Transmit() or
EFI_TCP4_PROTOCOL.Receive(). If NULL, all pending tokens
issued by above four functions will be aborted. Type
EFI_TCP4_COMPLETION_TOKEN is defined in
EFI_TCP4_PROTOCOL.Connect().

Description

The Cancel() function aborts a pending connection, listen, transmit or receive
request. If Token is not NULL and the token is in the connection, listen, transmission
or receive queue when it is being cancelled, its Token->Status will be set to
EFI_ABORTED and then Token->Event will be signaled. If the token is not in one of
the queues, which usually means that the asynchronous operation has completed,
EFI_NOT_FOUND is returned. If Token is NULL all asynchronous token issued by
Connect(), Accept(), Transmit() and Receive() will be aborted.

Status Codes Returned

<table>
<thead>
<tr>
<th>Code</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>EFI_SUCCESS</td>
<td>The asynchronous I/O request is aborted and Token-&gt;Event is signaled.</td>
</tr>
<tr>
<td>EFI_INVALID_PARAMETER</td>
<td>This is NULL.</td>
</tr>
<tr>
<td>EFI_NOT_STARTED</td>
<td>This instance hasn’t been configured.</td>
</tr>
<tr>
<td>EFI_NO_MAPPING</td>
<td>When using the default address, configuration (DHCP, BOOTP, RARP, etc.) hasn’t finished yet.</td>
</tr>
<tr>
<td>EFI_NOT_FOUND</td>
<td>The asynchronous I/O request isn’t found in the transmission or receive queue. It has either completed or wasn’t issued by Transmit() and Receive().</td>
</tr>
<tr>
<td>EFI_UNSUPPORTED</td>
<td>The implementation does not support this function.</td>
</tr>
</tbody>
</table>

EFI_TCP4_PROTOCOL.Poll()

Summary

Poll to receive incoming data and transmit outgoing segments.

Prototype

typedef

EFI_STATUS

(EIFIAPI *EFI_TCP4_POLL) (  
  IN EFI_TCP4_PROTOCOL  *This  );

Parameters

This Pointer to the EFI_TCP4_PROTOCOL instance.
Description

The \texttt{Poll()} function increases the rate that data is moved between the network and application and can be called when the TCP instance is created successfully. Its use is optional.

In some implementations, the periodical timer in the MNP driver may not poll the underlying communications device fast enough to avoid drop packets. Drivers and applications that are experiencing packet loss should try calling the \texttt{Poll()} function in a high frequency.

Status Codes Returned

<table>
<thead>
<tr>
<th>Status Code</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>\texttt{EFI_SUCCESS}</td>
<td>Incoming or outgoing data was processed.</td>
</tr>
<tr>
<td>\texttt{EFI_INVALID_PARAMETER}</td>
<td>This is \texttt{NULL}.</td>
</tr>
<tr>
<td>\texttt{EFI_DEVICE_ERROR}</td>
<td>An unexpected system or network error occurred.</td>
</tr>
<tr>
<td>\texttt{EFI_NOT_READY}</td>
<td>No incoming or outgoing data is processed.</td>
</tr>
<tr>
<td>\texttt{EFI_TIMEOUT}</td>
<td>Data was dropped out of the transmission or receive queue. Consider increasing the polling rate.</td>
</tr>
</tbody>
</table>

28.2 EFI TCPv6 Protocol

This section defines the EFI TCPv6 (Transmission Control Protocol version 6) Protocol.

28.2.1 TCPv6 Service Binding Protocol

\texttt{EFI_TCP6_SERVICE_BINDING_PROTOCOL}

Summary

The EFI TCPv6 Service Binding Protocol is used to locate EFI TCPv6 Protocol drivers to create and destroy protocol child instance of the driver to communicate with other host using TCP protocol.

GUID

\begin{verbatim}
#define EFI_TCP6_SERVICE_BINDING_PROTOCOL_GUID \
{0xec20eb79,0x6c1a,0x4664,} \
{0x9a,0x0d,0xd2,0xe4,0xcc,0x16,0xd6, 0x64}
\end{verbatim}

Description

A network application that requires TCPv6 I/O services can call one of the protocol handler services, such as \texttt{BS->LocateHandleBuffer()}, to search devices that publish an EFI TCPv6 Service Binding Protocol GUID. Such device supports the EFI TCPv6 Protocol and may be available for use.

After a successful call to the \texttt{EFI_TCP6_SERVICE_BINDING_PROTOCOL.CreateChild()} function, the newly created child EFI TCPv6 Protocol driver is in an un-configured state; it is not ready to do any operation except \texttt{Poll()} send and receive data packets until configured.
Every successful call to the `EFI_TCP6_SERVICE_BINDING_PROTOCOL.CreateChild()` function must be matched with a call to the `EFI_TCP6_SERVICE_BINDING_PROTOCOL.DestroyChild()` function to release the protocol driver.

### 28.2.2 TCPv6 Protocol

#### EFI_TCP6_PROTOCOL

**Summary**

The EFI TCPv6 Protocol provides services to send and receive data stream.

**GUID**

```c
#define EFI_TCP6_PROTOCOL_GUID {0x46e44855,0xbd60,0x4ab7,
{0xab,0x0d,0xa6,0x79,0xb9,0x44,0x7d,0x77}}
```

**Protocol Interface Structure**

```c
typedef struct _EFI_TCP6_PROTOCOL {
  EFI_TCP6_GET_MODE_DATA  GetModeData;
  EFI_TCP6_CONFIGURE      Configure;
  EFI_TCP6_CONNECT        Connect;
  EFI_TCP6_ACCEPT         Accept;
  EFI_TCP6_TRANSMIT       Transmit;
  EFI_TCP6_RECEIVE        Receive;
  EFI_TCP6_CLOSE          Close;
  EFI_TCP6_CANCEL         Cancel;
  EFI_TCP6_POLL           Poll;
} EFI_TCP6_PROTOCOL;
```

**Parameters**

- **GetModeData**
  
  Get the current operational status. See the `GetModeData()` function description.

- **Configure**
  
  Initialize, change, or brutally reset operational settings of the EFI TCPv6 Protocol. See the `Configure()` function description.

- **Connect**
  
  Initiate the TCP three-way handshake to connect to the remote peer configured in this TCP instance. The function is a nonblocking operation. See the `Connect()` function description.

- **Accept**
  
  Listen for incoming TCP connection requests. This function is a nonblocking operation. See the `Accept()` function description.
Transmit

Queue outgoing data to the transmit queue. This function is a nonblocking operation. See the Transmit() function description.

Receive

Queue a receiving request token to the receive queue. This function is a nonblocking operation. See the Receive() function description.

Close

Gracefully disconnect a TCP connection follow RFC 793 or reset a TCP connection. This function is a nonblocking operation. See the Close() function description.

Cancel

Abort a pending connect, listen, transmit or receive request. See the Cancel() function description.

Poll

Poll to receive incoming data and transmit outgoing TCP segments. See the Poll() function description.

Description

The EFI_TCP6_PROTOCOL defines the EFI TCPv6 Protocol child to be used by any network drivers or applications to send or receive data stream. It can either listen on a specified port as a service or actively connect to remote peer as a client. Each instance has its own independent settings.

Note: Byte Order: In this document, all IPv6 addresses and incoming/outgoing packets are stored in network byte order. All other parameters in the functions and data structures that are defined in this document are stored in host byte order unless explicitly specified.

EFI_TCP6_PROTOCOL.GetModeData()

Summary

Get the current operational status.

Prototype

typedef

EFI_STATUS

(EIFIAPI *EFI_TCP6_GET_MODE_DATA) (  
  IN EFI_TCP6_PROTOCOL        *This,  
  OUT EFI_TCP6_CONNECTION_STATE    *Tcp6State OPTIONAL,  
  OUT EFI_TCP6_CONFIG_DATA      *Tcp6ConfigData OPTIONAL,  
  OUT EFI_IPv6_MODE_DATA       *Ip6ModeData OPTIONAL,  
  OUT EFI_MANAGED_NETWORK_CONFIG_DATA *MnpConfigData OPTIONAL,  
  OUT EFI_SIMPLE_NETWORK_MODE     *SnpModeData OPTIONAL  
);

Parameters

This

Pointer to the EFI_TCP6_PROTOCOL instance.

Tcp6State

The buffer in which the current TCP state is returned. Type EFI_TCP6_CONNECTION_STATE is defined in "Related Definitions" below.
Tcp6ConfigData The buffer in which the current TCP configuration is returned. Type EFI_TCP6_CONFIG_DATA is defined in "Related Definitions" below.

Ip6ModeData The buffer in which the current IPv6 configuration data used by the TCP instance is returned. Type EFI_IP6_MODE_DATA is defined in EFI_IP6_PROTOCOL.GetModeData().

MnpConfigData The buffer in which the current MNP configuration data used indirectly by the TCP instance is returned. Type EFI_MANAGED_NETWORK_CONFIG_DATA is defined in EFI_MANAGED_NETWORK_PROTOCOL.GetModeData().

SnpModeData The buffer in which the current SNP mode data used indirectly by the TCP instance is returned. Type EFI_SIMPLE_NETWORK_MODE is defined in the EFI_SIMPLE_NETWORK_PROTOCOL.

Description

The GetModeData() function copies the current operational settings of this EFI TCPv6 Protocol instance into user-supplied buffers. This function can also be used to retrieve the operational setting of underlying drivers such as IPv6, MNP, or SNP.

Related Definition

typedef struct {
    EFI_IPv6_ADDRESS StationAddress;
    UINT16 StationPort;
    EFI_IPv6_ADDRESS RemoteAddress;
    UINT16 RemotePort;
    BOOLEAN ActiveFlag;
} EFI_TCP6_ACCESS_POINT;

StationAddress The local IP address assigned to this TCP instance. The EFI TCPv6 driver will only deliver incoming packets whose destination addresses exactly match the IP address. Set to zero to let the underlying IPv6 driver choose a source address. If not zero it must be one of the configured IP addresses in the underlying IPv6 driver.

StationPort The local port number to which this EFI TCPv6 Protocol instance is bound. If the instance doesn't care the local port number, set StationPort to zero to use an ephemeral port.

RemoteAddress The remote IP address to which this EFI TCPv6 Protocol instance is connected. If ActiveFlag is FALSE (i.e., a passive TCPv6 instance), the instance only accepts connections from the RemoteAddress. If ActiveFlag is TRUE the instance will connect to the RemoteAddress, i.e., outgoing segments will be sent to this address and only segments from this address will be delivered to the application. When ActiveFlag is FALSE, it
can be set to zero and means that incoming connection requests from any address will be accepted.

**RemotePort**
The remote port to which this EFI TCPv6 Protocol instance connects or from which connection request will be accepted by this EFI TCPv6 Protocol instance. If **ActiveFlag** is **FALSE** it can be zero and means that incoming connection request from any port will be accepted. Its value can not be zero when **ActiveFlag** is **TRUE**.

**ActiveFlag**
Set it to **TRUE** to initiate an active open. Set it to **FALSE** to initiate a passive open to act as a server.

---

```
//***********************************************************
// EFI_TCP6_OPTION
//***********************************************************
typedef struct {
    UINT32 ReceiveBufferSize;
    UINT32 SendBufferSize;
    UINT32 MaxSynBackLog;
    UINT32 ConnectionTimeout;
    UINT32 DataRetries;
    UINT32 FinTimeout;
    UINT32 TimeWaitTimeout;
    UINT32 KeepAliveProbes;
    UINT32 KeepAliveTime;
    UINT32 KeepAliveInterval;
    BOOLEAN EnableNagle;
    BOOLEAN EnableTimeStamp;
    BOOLEAN EnableWindowScaling;
    BOOLEAN EnableSelectiveAck;
    BOOLEAN EnablePathMtuDiscovery;
} EFI_TCP6_OPTION;
```

- **ReceiveBufferSize**: The size of the TCP receive buffer.
- **SendBufferSize**: The size of the TCP send buffer.
- **MaxSynBackLog**: The length of incoming connect request queue for a passive instance. When set to zero, the value is implementation specific.
- **ConnectionTimeout**: The maximum seconds a TCP instance will wait for before a TCP connection established. When set to zero, the value is implementation specific.
- **DataRetries**: The number of times TCP will attempt to retransmit a packet on an established connection. When set to zero, the value is implementation specific.
FinTimeout  How many seconds to wait in the FIN_WAIT_2 states for a final FIN flag before the TCP instance is closed. This timeout is in effective only if the application has called Close() to disconnect the connection completely. It is also called FIN_WAIT_2 timer in other implementations. When set to zero, it should be disabled because the FIN_WAIT_2 timer itself is against the standard.

TimeWaitTimeout  How many seconds to wait in TIME_WAIT state before the TCP instance is closed. The timer is disabled completely to provide a method to close the TCP connection quickly if it is set to zero. It is against the related RFC documents.

KeepAliveProbes  The maximum number of TCP keep-alive probes to send before giving up and resetting the connection if no response from the other end. Set to zero to disable keep-alive probe.

KeepAliveTime  The number of seconds a connection needs to be idle before TCP sends out periodical keep-alive probes. When set to zero, the value is implementation specific. It should be ignored if keep-alive probe is disabled.

KeepAliveInterval  The number of seconds between TCP keep-alive probes after the periodical keep-alive probe if no response. When set to zero, the value is implementation specific. It should be ignored if keep-alive probe is disabled.

EnableNagle  Set it to TRUE to enable the Nagle algorithm as defined in RFC896. Set it to FALSE to disable it.

EnableTimeStamp  Set it to TRUE to enable TCP timestamps option as defined in RFC7323. Set to FALSE to disable it.

EnableWindowScaling  Set it to TRUE to enable TCP window scale option as defined in RFC7323. Set it to FALSE to disable it.

EnableSelectiveAck  Set it to TRUE to enable selective acknowledge mechanism described in RFC 2018. Set it to FALSE to disable it. Implementation that supports SACK can optionally support DSAK as defined in RFC 2883.

EnablePathMtudiscovery  Set it to TRUE to enable path MTU discovery as defined in RFC 1191. Set to FALSE to disable it.

Option setting with digital value will be modified by driver if it is set out of the implementation specific range and an implementation specific default value will be set accordingly.
typedef struct {
    UINT8 TrafficClass;
    UINT8 HopLimit;
    EFI_TCP6_ACCESS_POINT AccessPoint;
    EFI_TCP6_OPTION *ControlOption;
} EFI_TCP6_CONFIG_DATA;

TrafficClass TrafficClass field in transmitted IPv6 packets.
HopLimit HopLimit field in transmitted IPv6 packets.
AccessPoint Used to specify TCP communication end settings for a TCP instance.
ControlOption Used to configure the advance TCP option for a connection. If set to NULL, implementation specific options for TCP connection will be used.

typedef enum {
    Tcp6StateClosed = 0,
    Tcp6StateListen = 1,
    Tcp6StateSynSent = 2,
    Tcp6StateSynReceived = 3,
    Tcp6StateEstablished = 4,
    Tcp6StateFinWait1 = 5,
    Tcp6StateFinWait2 = 6,
    Tcp6StateClosing = 7,
    Tcp6StateTimeWait = 8,
    Tcp6StateCloseWait = 9,
    Tcp6StateLastAck = 10
} EFI_TCP6_CONNECTION_STATE;

Status Codes Returned

<table>
<thead>
<tr>
<th>Status Code</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>EFI_SUCCESS</td>
<td>The mode data was read.</td>
</tr>
<tr>
<td>EFI_NOT_STARTED</td>
<td>No configuration data is available because this instance hasn’t been started.</td>
</tr>
<tr>
<td>EFI_INVALID_PARAMETER</td>
<td>This is NULL.</td>
</tr>
</tbody>
</table>
EFI_TCP6_PROTOCOL.Configure()

Summary
Initialize or brutally reset the operational parameters for this TCP instance.

Prototype

```
typedef EFI_STATUS (EFIAPI *EFI_TCP6_CONFIGURE) (IN EFI_TCP6_PROTOCOL *This,
IN EFI_TCP6_CONFIG_DATA *Tcp6ConfigData OPTIONAL);
```

Parameters

- **This**: Pointer to the EFI_TCP6_PROTOCOL instance.
- **Tcp6ConfigData**: Pointer to the configure data to configure the instance.

Description

The **Configure()** function does the following:

- Initialize this TCP instance, i.e., initialize the communication end settings and specify active open or passive open for an instance.
- Reset this TCP instance brutally, i.e., cancel all pending asynchronous tokens, flush transmission and receiving buffer directly without informing the communication peer.

No other TCPv6 Protocol operation except **Poll()** can be executed by this instance until it is configured properly. For an active TCP instance, after a proper configuration it may call **Connect()** to initiates the three-way handshake. For a passive TCP instance, its state will transit to **Tcp6StateListen** after configuration, and **Accept()** may be called to listen the incoming TCP connection requests. If **Tcp6ConfigData** is set to **NULL**, the instance is reset. Resetting process will be done brutally, the state machine will be set to **Tcp6StateClosed** directly, the receive queue and transmit queue will be flushed, and no traffic is allowed through this instance.

Status Codes Returned

<table>
<thead>
<tr>
<th>Code</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>EFI_SUCCESS</td>
<td>The operational settings are set, changed, or reset successfully.</td>
</tr>
<tr>
<td>EFI_NO_MAPPING</td>
<td>The underlying IPv6 driver was responsible for choosing a source address for this instance, but no source address was available for use.</td>
</tr>
</tbody>
</table>
**EFI_TCP6_PROTOCOL.Connect()**

**Summary**
Initiate a nonblocking TCP connection request for an active TCP instance.

**Prototype**
```c
typedef
EFI_STATUS
(EFIAPI *EFI_TCP6_CONNECT) (  
    IN EFI_TCP6_PROTOCOL *This,
    IN EFI_TCP6_CONNECTION_TOKEN *ConnectionToken
);
```

**Parameters**
- **This**: Pointer to the **EFI_TCP6_PROTOCOL** instance.
- **ConnectionToken**: Pointer to the connection token to return when the TCP three-way handshake finishes. Type **EFI_TCP6_CONNECTION_TOKEN** is defined in Related Definition below.

**Description**
The `Connect()` function will initiate an active open to the remote peer configured in current TCP instance if it is configured active. If the connection succeeds or fails due to any error, the `ConnectionToken->CompletionToken.Event` will be signaled and
ConnectionToken->CompletionToken.Status will be updated accordingly. This function can only be called for the TCP instance in Tcp6StateClosed state. The instance will transfer into Tcp6StateSynSent if the function returns EFI_SUCCESS. If TCP three-way handshake succeeds, its state will become Tcp6StateEstablished, otherwise, the state will return to Tcp6StateClosed.

Related Definitions

//***************************************************************
// EFI_TCP6_COMPLETION_TOKEN
//***************************************************************
typedef struct {
  EFI_EVENT Event;
  EFI_STATUS Status;
} EFI_TCP6_COMPLETION_TOKEN;

Event
The Event to signal after request is finished and Status field is updated by the EFI TCPv6 Protocol driver. The type of Event must be EVT_NOTIFY_SIGNAL.

Status
The result of the completed operation. EFI_NO_MEDIA. There was a media error

The EFI_TCP6_COMPLETION_TOKEN is used as a common header for various asynchronous tokens.

//***************************************************************
// EFI_TCP6_CONNECTION_TOKEN
//***************************************************************
typedef struct {
  EFI_TCP6_COMPLETION_TOKEN CompletionToken;
} EFI_TCP6_CONNECTION_TOKEN;

Status
The Status in the CompletionToken will be set to one of the following values if the active open succeeds or an unexpected error happens:

EFI_SUCCESS: The active open succeeds and the instance’s state is Tcp6StateEstablished.

EFI_CONNECTION_RESET: The connect fails because the connection is reset either by instance itself or the communication peer.

EFI_CONNECTION_REFUSED: The receiving or transmission operation fails because this connection is refused.

EFI_ABORTED: The active open is aborted.
**EFI_TIMEOUT**: The connection establishment timer expires and no more specific information is available.

**EFI_NETWORK_UNREACHABLE**: The active open fails because an ICMP network unreachable error is received.

**EFI_HOST_UNREACHABLE**: The active open fails because an ICMP host unreachable error is received.

**EFI_PROTOCOL_UNREACHABLE**: The active open fails because an ICMP protocol unreachable error is received.

**EFI_PORT_UNREACHABLE**: The connection establishment timer times out and an ICMP port unreachable error is received.

**EFI_ICMP_ERROR**: The connection establishment timer times out and some other ICMP error is received.

**EFI_DEVICE_ERROR**: An unexpected system or network error occurred.

**EFI_SECURITY_VIOLATION**: The active open was failed because of IPSec policy check.

### Status Codes Returned

<table>
<thead>
<tr>
<th>Code</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>EFI_SUCCESS</td>
<td>The connection request is successfully initiated and the state of this TCP instance has been changed to Tcp6StateSynSent.</td>
</tr>
<tr>
<td>EFI_NOT_STARTED</td>
<td>This EFI TCPv6 Protocol instance has not been configured.</td>
</tr>
<tr>
<td>EFI_ACCESS_DENIED</td>
<td>One or more of the following conditions are TRUE:</td>
</tr>
<tr>
<td></td>
<td>• This instance is not configured as an active one.</td>
</tr>
<tr>
<td></td>
<td>• This instance is not in Tcp6StateClosed state.</td>
</tr>
<tr>
<td>EFI_INVALID_PARAMETER</td>
<td>One or more of the following are TRUE:</td>
</tr>
<tr>
<td></td>
<td>• This is NULL.</td>
</tr>
<tr>
<td></td>
<td>• ConnectionToken is NULL.</td>
</tr>
<tr>
<td></td>
<td>• ConnectionToken-&gt;CompletionToken.Event is NULL.</td>
</tr>
<tr>
<td>EFI_OUT_OF_RESOURCES</td>
<td>The driver can’t allocate enough resource to initiate the active open.</td>
</tr>
<tr>
<td>EFI_DEVICE_ERROR</td>
<td>An unexpected system or network error occurred.</td>
</tr>
</tbody>
</table>

### EFI_TCP6_PROTOCOL.Accept()

**Summary**

Listen on the passive instance to accept an incoming connection request. This is a nonblocking operation.
Prototype

typedef
EFI_STATUS
(EIFIAPI *EFI_TCP6_ACCEPT) (  
    IN EFI_TCP6_PROTOCOL *This,
    IN EFI_TCP6_LISTEN_TOKEN *ListenToken
);

Parameters

This Pointer to the EFI_TCP6_PROTOCOL instance.

ListenToken Pointer to the listen token to return when operation finishes. Type EFI_TCP6_LISTEN_TOKEN is defined in Related Definition below.

Related Definitions

//***************************************************************
// EFI_TCP6_LISTEN_TOKEN
//***************************************************************
typedef struct {
    EFI_TCP6_COMPLETION_TOKEN CompletionToken;
    EFI_HANDLE NewChildHandle;
} EFI_TCP6_LISTEN_TOKEN;

Status The Status in CompletionToken will be set to the following value if accept finishes:

EFI_SUCCESS: A remote peer has successfully established a connection to this instance. A new TCP instance has also been created for the connection.

EFI_CONNECTION_RESET: The accept fails because the connection is reset either by instance itself or communication peer.

EFI_ABORTED: The accept request has been aborted.

EFI_SECURITY_VIOLATION: The accept operation was failed because of IPSec policy check.

NewChildHandle The new TCP instance handle created for the established connection.

Description

The Accept() function initiates an asynchronous accept request to wait for an incoming connection on the passive TCP instance. If a remote peer successfully establishes a connection with this instance, a new TCP instance will be created and its handle will be returned in ListenToken->NewChildHandle. The newly created instance is configured by inheriting the passive instance’s configuration and is ready for use upon return. The new instance is in the Tcp6StateEstablished state.
The `ListenToken->CompletionToken.Event` will be signaled when a new connection is accepted, user aborts the listen or connection is reset.

This function only can be called when current TCP instance is in `Tcp6StateListen` state.

### Status Codes Returned

<table>
<thead>
<tr>
<th>Status Code</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>EFI_SUCCESS</td>
<td>The listen token has been queued successfully.</td>
</tr>
<tr>
<td>EFI_NOT_STARTED</td>
<td>This EFI TCPv6 Protocol instance has not been configured.</td>
</tr>
<tr>
<td>EFI_ACCESS_DENIED</td>
<td>One or more of the following are <strong>TRUE</strong>:</td>
</tr>
<tr>
<td></td>
<td>• This instance is not a passive instance.</td>
</tr>
<tr>
<td></td>
<td>• This instance is not in <code>Tcp6StateListen</code> state.</td>
</tr>
<tr>
<td></td>
<td>• The same listen token has already existed in the listen token queue of this TCP instance.</td>
</tr>
<tr>
<td>EFI_INVALID_PARAMETER</td>
<td>One or more of the following are <strong>TRUE</strong>:</td>
</tr>
<tr>
<td></td>
<td>• This is <strong>NULL</strong>.</td>
</tr>
<tr>
<td></td>
<td>• <code>ListenToken</code> is <strong>NULL</strong>.</td>
</tr>
<tr>
<td></td>
<td>• <code>ListenToken-&gt;CompletionToken.Event</code> is <strong>NULL</strong>.</td>
</tr>
<tr>
<td>EFI_OUT_OF_RESOURCES</td>
<td>Could not allocate enough resource to finish the operation.</td>
</tr>
<tr>
<td>EFI_DEVICE_ERROR</td>
<td>Any unexpected and not belonged to above category error.</td>
</tr>
</tbody>
</table>

### EFI_TCP6_PROTOCOL.Transmit()

#### Summary

Queues outgoing data into the transmit queue.

#### Prototype

```c
typedef
    EFI_STATUS
    (EFIAPI *EFI_TCP6_TRANSMIT) (
        IN EFI_TCP6_PROTOCOL      *This,
        IN EFI_TCP6_IO_TOKEN      *Token
    );
```

#### Parameters

- **This**
  - Pointer to the `EFI_TCP6_PROTOCOL` instance.
- **Token**
  - Pointer to the completion token to queue to the transmit queue. Type `EFI_TCP6_IO_TOKEN` is defined in "Related Definitions" below.

#### Description

The `Transmit()` function queues a sending request to this TCP instance along with the user data.
The status of the token is updated and the event in the token will be signaled once the data is sent out or some error occurs.

Related Definitions

```c
typedef struct {
    EFI_TCP6_COMPLETION_TOKEN CompletionToken;
    union {
        EFI_TCP6_RECEIVE_DATA   *RxData;
        EFI_TCP6_TRANSMIT_DATA  *TxData;
    }
    Packet;
} EFI_TCP6_IO_TOKEN;
```

**Status**

When transmission finishes or meets any unexpected error it will be set to one of the following values:

- **EFI_SUCCESS**: The receiving or transmission operation completes successfully.
- **EFI_CONNECTION_FIN**: The receiving operation fails because the communication peer has closed the connection and there is no more data in the receive buffer of the instance.
- **EFI_CONNECTION_RESET**: The receiving or transmission operation fails because this connection is reset either by instance itself or the communication peer.
- **EFI_ABORTED**: The receiving or transmission is aborted.
- **EFI_TIMEOUT**: The transmission timer expires and no more specific information is available.
- **EFI_NETWORK_UNREACHABLE**: The transmission fails because an ICMP network unreachable error is received.
- **EFI_HOST_UNREACHABLE**: The transmission fails because an ICMP host unreachable error is received.
- **EFI_PROTOCOL_UNREACHABLE**: The transmission fails because an ICMP protocol unreachable error is received.
- **EFI_PORT_UNREACHABLE**: The transmission fails and an ICMP port unreachable error is received.
- **EFI_ICMP_ERROR**: The transmission fails and some other ICMP error is received.
- **EFI_DEVICE_ERROR**: An unexpected system or network error occurs.
- **EFI_SECURITY_VIOLATION**: The receiving or transmission operation was failed because of IPSec policy check.
When this token is used for receiving, RxData is a pointer to EFI_TCP6_RECEIVE_DATA. Type EFI_TCP6_RECEIVE_DATA is defined below.

When this token is used for transmitting, TxData is a pointer to EFI_TCP6_TRANSMIT_DATA. Type EFI_TCP6_TRANSMIT_DATA is defined below.

The EFI_TCP6_IO_TOKEN structure is used for both transmit and receive operations.

When used for transmitting, the CompletionToken.Event and TxData fields must be filled in by the user. After the transmit operation completes, the CompletionToken.Status field is updated by the instance and the Event is signaled.

When used for receiving, the CompletionToken.Event and RxData fields must be filled in by the user. After a receive operation completes, RxData and Status are updated by the instance and the Event is signaled.

```c
typedef struct {
  BOOLEAN UrgentFlag;
  UINT32 DataLength;
  UINT32 FragmentCount;
  EFI_TCP6_FRAGMENT_DATA FragmentTable[1];
} EFI_TCP6_RECEIVE_DATA;
```

UrgentFlag Whether the data is urgent. When this flag is set, the instance is in urgent mode. The implementations of this specification should follow RFC793 to process urgent data, and should NOT mix the data across the urgent point in one token.

DataLength When calling Receive() function, it is the byte counts of all FragmentBuffer in FragmentTable allocated by user. When the token is signaled by TCPv6 driver it is the length of received data in the fragments.

FragmentCount Number of fragments.

FragmentTable An array of fragment descriptors. Type EFI_TCP6_FRAGMENT_DATA is defined below.

When TCPv6 driver wants to deliver received data to the application, it will pick up the first queued receiving token, update its Token->Packet.RxData then signal the Token->CompletionToken.Event.

The FragmentBuffer in FragmentTable is allocated by the application when calling Receive() function and received data will be copied to those buffers by the driver. FragmentTable may contain multiple buffers that are NOT in the continuous
memory locations. The application should combine those buffers in the FragmentTable to process data if necessary.

```c
//****************************************************************************
// EFI_TCP6_FRAGMENT_DATA
//*****************************************************************************
typedef struct {
  UINT32 FragmentLength;
  VOID *FragmentBuffer;
} EFI_TCP6_FRAGMENT_DATA;
```

- **FragmentLength**: Length of data buffer in the fragment.
- **FragmentBuffer**: Pointer to the data buffer in the fragment.

**EFI_TCP6_FRAGMENT_DATA** allows multiple receive or transmit buffers to be specified. The purpose of this structure is to provide scattered read and write.

```c
//****************************************************************************
// EFI_TCP6_TRANSMIT_DATA
//*****************************************************************************
typedef struct {
  BOOLEAN Push;
  BOOLEAN Urgent;
  UINT32 DataLength;
  UINT32 FragmentCount;
  EFI_TCP6_FRAGMENT_DATA FragmentTable[1];
} EFI_TCP6_TRANSMIT_DATA;
```

- **Push**: If **TRUE**, data must be transmitted promptly, and the PUSH bit in the last TCP segment created will be set. If **FALSE**, data transmission may be delayed to combine with data from subsequent Transmit()s for efficiency.
- **Urgent**: The data in the fragment table are urgent and urgent point is in effect if **TRUE**. Otherwise those data are NOT considered urgent.
- **DataLength**: Length of the data in the fragments.
- **FragmentCount**: Number of fragments.
- **FragmentTable**: An array of fragment descriptors. Type **EFI_TCP6_FRAGMENT_DATA** is defined above.
The EFI TCPv6 Protocol user must fill this data structure before sending a packet. The packet may contain multiple buffers in non-continuous memory locations.

### Status Codes Returned

<table>
<thead>
<tr>
<th>Code</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>EFI_SUCCESS</td>
<td>The data has been queued for transmission.</td>
</tr>
<tr>
<td>EFI_NOT_STARTED</td>
<td>This EFI TCPv6 Protocol instance has not been configured.</td>
</tr>
<tr>
<td>EFI_NO_MAPPING</td>
<td>The underlying IPv6 driver was responsible for choosing a source address for this instance, but no source address was available for use.</td>
</tr>
</tbody>
</table>
| EFI_INVALID_PARAMETER | One or more of the following are **TRUE**:  
  - This is **NULL**  
  - Token is **NULL**  
  - Token->CompletionToken.Event is **NULL**  
  - Token->Packet.TxData is **NULL**  
  - Token->Packet.FragmentCount is zero.  
  - Token->Packet.DataLength is not equal to the sum of fragment lengths. |
| EFI_ACCESS_DENIED  | One or more of the following conditions are **TRUE**:  
  - A transmit completion token with the same Token->CompletionToken.Event was already in the transmission queue.  
  - The current instance is in Tcp6StateClosed state.  
  - The current instance is a passive one and it is in Tcp6StateListen state.  
  - User has called **Close()** to disconnect this connection. |
| EFI_NOT_READY      | The completion token could not be queued because the transmit queue is full. |
| EFI_OUT_OF_RESOURCES | Could not queue the transmit data because of resource shortage. |
| EFI_NETWORK_UNREACHABLE | There is no route to the destination network or address. |
| EFI_NO_MEDIA       | There was a media error.                                                    |

### EFI_TCP6_PROTOCOL.Receive()

**Summary**

Places an asynchronous receive request into the receiving queue.

**Prototype**

```c
typedef
EFI_STATUS
(EIFIAPI *EFI_TCP6_RECEIVE) (  
  IN EFI_TCP6_PROTOCOL     *This,
  IN EFI_TCP6_IO_TOKEN     *Token
 );
```
Parameters

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>This</td>
<td>Pointer to the EFI_TCP6_PROTOCOL instance.</td>
</tr>
<tr>
<td>Token</td>
<td>Pointer to a token that is associated with the receive data descriptor. Type EFI_TCP6_IO_TOKEN is defined in EFI_TCP6_PROTOCOL.Transmit().</td>
</tr>
</tbody>
</table>

Description

The Receive() function places a completion token into the receive packet queue. This function is always asynchronous. The caller must allocate the Token->CompletionToken.Event and the FragmentBuffer used to receive data. The caller also must fill the DataLength which represents the whole length of all FragmentBuffer. When the receive operation completes, the EFI TCPv6 Protocol driver updates the Token->CompletionToken.Status and Token->Packet.RxData fields and the Token->CompletionToken.Event is signaled. If got data the data and its length will be copied into the FragmentTable, at the same time the full length of received data will be recorded in the DataLength fields. Providing a proper notification function and context for the event will enable the user to receive the notification and receiving status. That notification function is guaranteed to not be re-entered.

Status Codes Returned

<table>
<thead>
<tr>
<th>Code</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>EFI_SUCCESS</td>
<td>The receive completion token was cached.</td>
</tr>
<tr>
<td>EFI_NOT_STARTED</td>
<td>This EFI TCPv6 Protocol instance has not been configured.</td>
</tr>
<tr>
<td>EFI_NO_MAPPING</td>
<td>The underlying IPv6 driver was responsible for choosing a source address for this instance, but no source address was available for use.</td>
</tr>
</tbody>
</table>
| EFI_INVALID_PARAMETER | One or more of the following conditions is TRUE:  
  • This is NULL.  
  • Token is NULL.  
  • Token->CompletionToken.Event is NULL.  
  • Token->Packet.RxData is NULL.  
  • Token->Packet.RxData->DataLength is 0.  
  • The Token->Packet.RxData->DataLength is not the sum of all FragmentBuffer length in FragmentTable. |
| EFI_OUT_OF_RESOURCES | The receive completion token could not be queued due to a lack of system resources (usually memory). |
| EFI_DEVICE_ERROR | An unexpected system or network error occurred. The EFI TCPv6 Protocol instance has been reset to startup defaults. |
| EFI_ACCESS_DENIED | One or more of the following conditions is TRUE:  
  • A receive completion token with the same Token->CompletionToken.Event was already in the receive queue.  
  • The current instance is in Tcp6StateClosed state.  
  • The current instance is a passive one and it is in Tcp6StateListen state.  
  • User has called Close() to disconnect this connection. |
EFI_TCP6_PROTOCOL.Close()

Summary
Disconnecting a TCP connection gracefully or reset a TCP connection. This function is a nonblocking operation.

Prototype
```c
typedef EFI_STATUS (EFIAPI *EFI_TCP6_CLOSE)(
    IN EFI_TCP6_PROTOCOL *This,
    IN EFI_TCP6_CLOSE_TOKEN *CloseToken);
```

Parameters
- **This**: Pointer to the EFI_TCP6_PROTOCOL instance.
- **CloseToken**: Pointer to the close token to return when operation finishes. Type EFI_TCP6_CLOSE_TOKEN is defined in Related Definition below.

Related Definitions
```c
//***************************************************************
// EFI_TCP6_CLOSE_TOKEN
//***************************************************************
typedef struct {
    EFI_TCP6_COMPLETION_TOKEN CompletionToken;
    BOOLEAN AbortOnClose;
} EFI_TCP6_CLOSE_TOKEN;
```

Status
When close finishes or meets any unexpected error it will be set to one of the following values:
- **EFI_SUCCESS**: The close operation completes successfully.
- **EFI_ABORTED**: User called configure with NULL without close stopping.
- **EFI_SECURITY_VIOLATION**: The close operation was failed because of IPSec policy check
- **EFI_CONNECTION_FIN**: The communication peer has closed the connection and there is no any buffered data in the receive buffer of this instance.
- **EFI_NOT_READY**: The receive request could not be queued because the receive queue is full.
- **EFI_NO_MEDIA**: There was a media error.

AbortOnClose
Abort the TCP connection on close instead of the standard TCP close process when it is set to **TRUE**. This option can be used to satisfy a fast disconnect.
**Description**

Initiate an asynchronous close token to TCP driver. After `Close()` is called, any buffered transmission data will be sent by TCP driver and the current instance will have a graceful close working flow described as RFC 793 if `AbortOnClose` is set to `FALSE`, otherwise, a rest packet will be sent by TCP driver to fast disconnect this connection. When the close operation completes successfully the TCP instance is in `Tcp6StateClosed` state, all pending asynchronous operations are signaled and any buffers used for TCP network traffic are flushed.

**Status Codes Returned**

<table>
<thead>
<tr>
<th>Code</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>EFI_SUCCESS</td>
<td>The <code>Close()</code> is called successfully.</td>
</tr>
<tr>
<td>EFI_NOT_STARTED</td>
<td>This EFI TCPv6 Protocol instance has not been configured.</td>
</tr>
<tr>
<td>EFI_ACCESS_DENIED</td>
<td>One or more of the following conditions are <code>TRUE</code>:</td>
</tr>
<tr>
<td></td>
<td>• <code>CloseToken</code> or <code>CloseToken-&gt;CompletionToken.Event</code> is already in use.</td>
</tr>
<tr>
<td></td>
<td>• Previous <code>Close()</code> call on this instance has not finished.</td>
</tr>
<tr>
<td>EFI_INVALID_PARAMETER</td>
<td>One or more of the following conditions are <code>TRUE</code>:</td>
</tr>
<tr>
<td></td>
<td>• <code>This</code> is <code>NULL</code>.</td>
</tr>
<tr>
<td></td>
<td>• <code>CloseToken</code> is <code>NULL</code>.</td>
</tr>
<tr>
<td></td>
<td>• <code>CloseToken-&gt;CompletionToken.Event</code> is <code>NULL</code>.</td>
</tr>
<tr>
<td>EFI_OUT_OF_RESOURCES</td>
<td>Could not allocate enough resource to finish the operation.</td>
</tr>
<tr>
<td>EFI_DEVICE_ERROR</td>
<td>Any unexpected and not belonged to above category error.</td>
</tr>
</tbody>
</table>

**EFI_TCP6_PROTOCOL.Cancel()**

**Summary**

Abort an asynchronous connection, listen, transmission or receive request.

**Prototype**

```c
typedef EFI_STATUS
(EFI_API *EFI_TCP6_CANCEL)(
    IN EFI_TCP6_PROTOCOL   *This,
    IN EFI_TCP6_COMPLETION_TOKEN  *Token  OPTIONAL
);
```

**Parameters**

- **This** Pointer to the `EFI_TCP6_PROTOCOL` instance.
- **Token** Pointer to a token that has been issued by `EFI_TCP6_PROTOCOL.Connect()`, `EFI_TCP6_PROTOCOL.Accept()`, `EFI_TCP6_PROTOCOL.Transmit()` or `EFI_TCP6_PROTOCOL.Receive()`. If `NULL`, all pending tokens issued by above four functions will be aborted. Type
EFI_TCP6_COMPLETION_TOKEN is defined in EFI_TCP6_PROTOCOL.Connect().

Description
The Cancel() function aborts a pending connection, listen, transmit or receive request. If Token is not NULL and the token is in the connection, listen, transmission or receive queue when it is being cancelled, its Token->Status will be set to EFI_ABORTED and then Token->Event will be signaled. If the token is not in one of the queues, which usually means that the asynchronous operation has completed, EFI_NOT_FOUND is returned. If Token is NULL all asynchronous token issued by Connect(), Accept(), Transmit() and Receive() will be aborted.

Status Codes Returned

<table>
<thead>
<tr>
<th>Code</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>EFI_SUCCESS</td>
<td>The asynchronous I/O request is aborted and Token-&gt;Event is signaled.</td>
</tr>
<tr>
<td>EFI_INVALID_PARAMETER</td>
<td>This is NULL.</td>
</tr>
<tr>
<td>EFI_NOT_STARTED</td>
<td>This instance hasn’t been configured.</td>
</tr>
<tr>
<td>EFI_NOT_FOUND</td>
<td>The asynchronous I/O request isn’t found in the transmission or receive queue. It has either completed or wasn’t issued by Transmit() and Receive().</td>
</tr>
<tr>
<td>EFI_UNSUPPORTED</td>
<td>The implementation does not support this function.</td>
</tr>
</tbody>
</table>

EFI_TCP6_PROTOCOL.Poll()

Summary
Poll to receive incoming data and transmit outgoing segments.

Prototype
typedef
EFI_STATUS
(EFI_API *EFI_TCP6_POLL) ( |
    IN EFI_TCP6_PROTOCOL   *This |
);

Parameters
This Pointer to the EFI_TCP6_PROTOCOL instance.

Description
The Poll() function increases the rate that data is moved between the network and application and can be called when the TCP instance is created successfully. Its use is optional.

In some implementations, the periodical timer in the MNP driver may not poll the underlying communications device fast enough to avoid drop packets. Drivers and applications that are experiencing packet loss should try calling the Poll() function in a high frequency.
Status Codes Returned

<table>
<thead>
<tr>
<th>Code</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>EFI_SUCCESS</td>
<td>Incoming or outgoing data was processed.</td>
</tr>
<tr>
<td>EFI_INVALID_PARAMETER</td>
<td>This is NULL.</td>
</tr>
<tr>
<td>EFI_DEVICE_ERROR</td>
<td>An unexpected system or network error occurred.</td>
</tr>
<tr>
<td>EFI_NOT_READY</td>
<td>No incoming or outgoing data is processed.</td>
</tr>
<tr>
<td>EFI_TIMEOUT</td>
<td>Data was dropped out of the transmission or receive queue. Consider increasing the polling rate.</td>
</tr>
</tbody>
</table>

28.3 EFI IPv4 Protocol

This section defines the EFI IPv4 (Internet Protocol version 4) Protocol interface. It is split into the following three main sections:

- EFI IPv4 Service Binding Protocol
- EFI IPv4 Variable
- EFI IPv4 Protocol

The EFI IPv4 Protocol provides basic network IPv4 packet I/O services, which includes support for a subset of the Internet Control Message Protocol (ICMP) and may include support for the Internet Group Management Protocol (IGMP).

The EFI IPv4 Protocol supports IPv4 classless IP addressing, and deprecates the original IPv4 classful IP addressing. Please see links to the following RFC documents at http://uefi.org/uefi:

1. RFC 1122 — “Requirements for Internet Hosts -- Communication Layers”
2. RFC 4632 — “Classless Inter-domain Routing (CIDR): The Internet Address Assignment and Aggregation Plan”
3. RFC 3021 — “Using 31-Bit Prefixes on IPv4 Point-to-Point Links”

28.3.1 IP4 Service Binding Protocol

**EFI_IP4_SERVICE_BINDING_PROTOCOL**

Summary

The EFI IPv4 Service Binding Protocol is used to locate communication devices that are supported by an EFI IPv4 Protocol driver and to create and destroy instances of the EFI IPv4 Protocol child protocol driver that can use the underlying communications device.

GUID

```c
#define EFI_IP4_SERVICE_BINDING_PROTOCOL_GUID
{0xc51711e7,0xb4bf,0x404a,
 {0xbf,0xb8,0x0a,0x04,0x8e,0xf1,0xff,0xe4}}
```

Description

A network application that requires basic IPv4 I/O services can use one of the protocol handler services, such as `BS->LocateHandleBuffer()`, to search for devices that publish an EFI IPv4 Service Binding Protocol GUID. Each device with a
published EFI IPv4 Service Binding Protocol GUID supports the EFI IPv4 Protocol and may be available for use.

After a successful call to the `EFI_IP4_SERVICE_BINDING_PROTOCOL.CreateChild()` function, the newly created child EFI IPv4 Protocol driver is in an unconfigured state; it is not ready to send and receive data packets.

Before a network application terminates execution, every successful call to the `EFI_IP4_SERVICE_BINDING_PROTOCOL.CreateChild()` function must be matched with a call to the `EFI_IP4_SERVICE_BINDING_PROTOCOL.DestroyChild()` function.

### 28.3.2 IP4 Protocol

**EFI_IP4_PROTOCOL**

**Summary**

The EFI IPv4 Protocol implements a simple packet-oriented interface that can be used by drivers, daemons, and applications to transmit and receive network packets.

**GUID**

```c
#define EFI_IP4_PROTOCOL_GUID \
{0x41d94cd2,0x35b6,0x455a,\ 
{0x82,0x58,0xd4,0xe5,0x13,0x34,0xaa,0xdd}
```

**Protocol Interface Structure**

```c
typedef struct _EFI_IP4_PROTOCOL {
  EFI_IP4_GET_MODE_DATA   GetModeData;
  EFI_IP4_CONFIGURE       Configure;
  EFI_IP4_GROUPS          Groups;
  EFI_IP4_ROUTES          Routes;
  EFI_IP4_TRANSMIT        Transmit;
  EFI_IP4_RECEIVE         Receive;
  EFI_IP4_CANCEL          Cancel;
  EFI_IP4_POLL            Poll;
} EFI_IP4_PROTOCOL;
```

**Parameters**

- **GetModeData**
  Gets the current operational settings for this instance of the EFI IPv4 Protocol driver. See the `GetModeData()` function description.

- **Configure**
  Changes or resets the operational settings for the EFI IPv4 Protocol. See the `Configure()` function description.

- **Groups**
  Joins and leaves multicast groups. See the `Groups()` function description.

- **Routes**
  Adds and deletes routing table entries. See the `Routes()` function description.
Transmit  Places outgoing data packets into the transmit queue. See the \texttt{Transmit()} function description.

Receive  Places a receiving request into the receiving queue. See the \texttt{Receive()} function description.

Cancel  Aborts a pending transmit or receive request. See the \texttt{Cancel()} function description.

Poll  Polls for incoming data packets and processes outgoing data packets. See the \texttt{Poll()} function description.

\textbf{Description}

The \texttt{EFI\_IP4\_PROTOCOL} defines a set of simple IPv4, ICMPv4, and IGMPv4 services that can be used by any network protocol driver, daemon, or application to transmit and receive IPv4 data packets.

\textbf{Note:} All the IPv4 addresses that are described in \texttt{EFI\_IP4\_PROTOCOL} are stored in network byte order. Both incoming and outgoing IP packets are also in network byte order. All other parameters that are defined in functions or data structures are stored in host byte order.

\texttt{EFI\_IP4\_PROTOCOL}.GetModeData()

\textbf{Summary}

Gets the current operational settings for this instance of the EFI IPv4 Protocol driver.

\textbf{Prototype}

\begin{verbatim}
typedef EFI_STATUS
(EFI_API *EFI_IP4_GET_MODE_DATA) (    
    IN EFI_IP4_PROTOCOL *This,
    OUT EFI_IP4_MODE_DATA *Ip4ModeData OPTIONAL,
    OUT EFI_MANAGED_NETWORK_CONFIG_DATA *MnpConfigData OPTIONAL,
    OUT EFI_SIMPLE_NETWORK_MODE *SnpData OPTIONAL
    );
\end{verbatim}

\textbf{Parameters}

\begin{itemize}
    \item \texttt{This} Pointer to the \texttt{EFI\_IP4\_PROTOCOL} instance.
    \item \texttt{Ip4ModeData} Pointer to the EFI IPv4 Protocol mode data structure. Type \texttt{EFI\_IP4\_MODE\_DATA} is defined in "Related Definitions" below.
    \item \texttt{MnpConfigData} Pointer to the managed network configuration data structure. Type \texttt{EFI\_MANAGED\_NETWORK\_CONFIG\_DATA} is defined in \texttt{EFI\_MANAGED\_NETWORK\_PROTOCOL}.\texttt{GetModeData()}.\n    \item \texttt{SnpData} Pointer to the simple network mode data structure. Type \texttt{EFI\_SIMPLE\_NETWORK\_MODE} is defined in the \texttt{EFI\_SIMPLE\_NETWORK\_PROTOCOL}.
**Description**

The `GetModeData()` function returns the current operational mode data for this driver instance. The data fields in `EFI_IP4_MODE_DATA` are read only. This function is used optionally to retrieve the operational mode data of underlying networks or drivers.

**Related Definitions**

```c
typedef struct {  
  BOOLEAN    IsStarted;  
  UINT32     MaxPacketSize;  
  EFI_IP4_CONFIG_DATA ConfigData;  
  BOOLEAN    IsConfigured;  
  UINT32     GroupCount;  
  EFI_IPv4_ADDRESS *GroupTable;  
  UINT32     RouteCount;  
  EFI_IP4_ROUTE_TABLE *RouteTable;  
  UINT32     IcmpTypeCount;  
  EFI_IP4_ICMP_TYPE *IcmpTypeList;  
} EFI_IP4_MODE_DATA;
```

- **IsStarted**: Set to `TRUE` after this EFI IPv4 Protocol instance has been successfully configured with operational parameters by calling the `Configure()` interface when EFI IPv4 Protocol instance is stopped. All other fields in this structure are undefined until this field is `TRUE`. Set to `FALSE` when the instance's operational parameter has been reset.

- **MaxPacketSize**: The maximum packet size, in bytes, of the packet which the upper layer driver could feed.

- **ConfigData**: Current configuration settings. Undefined until `IsStarted` is `TRUE`. Type `EFI_IP4_CONFIG_DATA` is defined below.

- **IsConfigured**: Set to `TRUE` when the EFI IPv4 Protocol instance has a station address and subnet mask. If it is using the default address, the default address has been acquired. Set to `FALSE` when the EFI IPv4 Protocol driver is not configured.

- **GroupCount**: Number of joined multicast groups. Undefined until `IsConfigured` is `TRUE`.

- **GroupTable**: List of joined multicast group addresses. Undefined until `IsConfigured` is `TRUE`.

- **RouteCount**: Number of entries in the routing table. Undefined until `IsConfigured` is `TRUE`. 

RouteTable          Routing table entries. Undefined until IsConfigured is TRUE. Type EFI_IP4_ROUTE_TABLE is defined below.
IcmpTypeCount       Number of entries in the supported ICMP types list.
IcmpTypeList        Array of ICMP types and codes that are supported by this EFI IPv4 Protocol driver. Type EFI_IP4_ICMP_TYPE is defined below.

The EFI_IP4_MODE_DATA structure describes the operational state of this IPv4 interface.

//******************************************************************************************
// EFI_IP4_CONFIG_DATA
//******************************************************************************************
typedef struct {
    UINT8     DefaultProtocol;
    BOOLEAN   AcceptAnyProtocol;
    BOOLEAN   AcceptIcmpErrors;
    BOOLEAN   AcceptBroadcast;
    BOOLEAN   AcceptPromiscuous;
    BOOLEAN   UseDefaultAddress;
    EFI_IPv4_ADDRESS  StationAddress;
    EFI_IPv4_ADDRESS  SubnetMask;
    UINT8     TypeOfService;
    UINT8     TimeToLive;
    BOOLEAN   DoNotFragment;
    BOOLEAN   RawData;
    UINT32    ReceiveTimeout;
    UINT32    TransmitTimeout;
} EFI_IP4_CONFIG_DATA;

DefaultProtocol     The default IPv4 protocol packets to send and receive. Ignored when AcceptPromiscuous is TRUE. An updated list of protocol numbers can be found at “Links to UEFI-Related Documents” (http://uefi.org/uefi) under the heading “IANA Assigned Internet Protocol Numbers list”.
AcceptAnyProtocol   Set to TRUE to receive all IPv4 packets that get through the receive filters. Set to FALSE to receive only the DefaultProtocol IPv4 packets that get through the receive filters. Ignored when AcceptPromiscuous is TRUE.
AcceptIcmpErrors    Set to TRUE to receive ICMP error report packets. Ignored when AcceptPromiscuous or AcceptAnyProtocol is TRUE.
AcceptBroadcast     Set to TRUE to receive broadcast IPv4 packets. Ignored when AcceptPromiscuous is TRUE. Set to FALSE to stop receiving broadcast IPv4 packets.
<table>
<thead>
<tr>
<th>Feature</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>AcceptPromiscuous</strong></td>
<td>Set to <strong>TRUE</strong> to receive all IPv4 packets that are sent to any hardware address or any protocol address. Set to <strong>FALSE</strong> to stop receiving all promiscuous IPv4 packets.</td>
</tr>
<tr>
<td><strong>UseDefaultAddress</strong></td>
<td>Set to <strong>TRUE</strong> to use the default IPv4 address and default routing table. If the default IPv4 address is not available yet, then the EFI IPv4 Protocol driver will use <strong>EFI_IP4_CONFIG2_PROTOCOL</strong> to retrieve the IPv4 address and subnet information. (This field can be set and changed only when the EFI IPv4 driver is transitioning from the stopped to the started states.)</td>
</tr>
<tr>
<td><strong>StationAddress</strong></td>
<td>The station IPv4 address that will be assigned to this EFI IPv4Protocol instance. The EFI IPv4 Protocol driver will deliver only incoming IPv4 packets whose destination matches this IPv4 address exactly. Address 0.0.0.0 is also accepted as a special case in which incoming packets destined to any station IP address are always delivered. When <strong>EFI_IP4_CONFIG_DATA</strong> is used in <strong>Configure ()</strong>, it is ignored if <strong>UseDefaultAddress</strong> is <strong>TRUE</strong>; When <strong>EFI_IP4_CONFIG_DATA</strong> is used in <strong>GetModeData ()</strong>, it contains the default address if <strong>UseDefaultAddress</strong> is <strong>TRUE</strong> and the default address has been acquired.</td>
</tr>
<tr>
<td><strong>SubnetMask</strong></td>
<td>The subnet address mask that is associated with the station address. When <strong>EFI_IP4_CONFIG_DATA</strong> is used in <strong>Configure ()</strong>, it is ignored if <strong>UseDefaultAddress</strong> is <strong>TRUE</strong>; When <strong>EFI_IP4_CONFIG_DATA</strong> is used in <strong>GetModeData ()</strong>, it contains the default subnet mask if <strong>UseDefaultAddress</strong> is <strong>TRUE</strong> and the default address has been acquired.</td>
</tr>
<tr>
<td><strong>TypeOfService</strong></td>
<td><strong>TypeOfService</strong> field in transmitted IPv4 packets.</td>
</tr>
<tr>
<td><strong>TimeToLive</strong></td>
<td><strong>TimeToLive</strong> field in transmitted IPv4 packets.</td>
</tr>
<tr>
<td><strong>DoNotFragment</strong></td>
<td>State of the <strong>DoNotFragment</strong> bit in transmitted IPv4 packets.</td>
</tr>
<tr>
<td><strong>RawData</strong></td>
<td>Set to <strong>TRUE</strong> to send and receive unformatted packets. The other IPv4 receive filters are still applied. Fragmentation is disabled for <strong>RawData</strong> mode. NOTE: Unformatted packets include the IP header and payload. The media header is appended automatically for outgoing packets by underlying network drivers.</td>
</tr>
<tr>
<td><strong>ReceiveTimeout</strong></td>
<td>The timer timeout value (number of microseconds) for the receive timeout event to be associated with each assembled packet. Zero means do not drop assembled packets.</td>
</tr>
<tr>
<td><strong>TransmitTimeout</strong></td>
<td>The timer timeout value (number of microseconds) for the transmit timeout event to be associated with each outgoing packet. Zero means do not drop outgoing packets.</td>
</tr>
</tbody>
</table>

The **EFI_IP4_CONFIG_DATA** structure is used to report and change IPv4 session parameters.
typedef struct {
    EFI_IPv4_ADDRESS SubnetAddress;
    EFI_IPv4_ADDRESS SubnetMask;
    EFI_IPv4_ADDRESS GatewayAddress;
} EFI_IP4_ROUTE_TABLE;

SubnetAddress  The subnet address to be routed.
SubnetMask     The subnet mask. If (DestinationAddress & SubnetMask == SubnetAddress), then the packet is to be directed to the GatewayAddress.
GatewayAddress The IPv4 address of the gateway that redirects packets to this subnet. If the IPv4 address is 0.0.0.0, then packets to this subnet are not redirected.

EFI_IP4_ROUTE_TABLE is the entry structure that is used in routing tables.

typedef struct {
    UINT8     Type;
    UINT8     Code;
} EFI_IP4_ICMP_TYPE

Type         The type of ICMP message. See RFC 792 and RFC 950.
Code         The code of the ICMP message, which further describes the different ICMP message formats under the same Type. See RFC 792 and RFC 950.

EFI_IP4_ICMP_TYPE is used to describe those ICMP messages that are supported by this EFI IPv4 Protocol driver.

Status Codes Returned

<table>
<thead>
<tr>
<th>Code</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>EFI_SUCCESS</td>
<td>The operation completed successfully.</td>
</tr>
<tr>
<td>EFI_INVALID_PARAMETER</td>
<td>This is NULL.</td>
</tr>
<tr>
<td>EFI_OUT_OF_RESOURCES</td>
<td>The required mode data could not be allocated.</td>
</tr>
</tbody>
</table>
EFI_IP4_PROTOCOL.Configure()

Summary
Assigns an IPv4 address and subnet mask to this EFI IPv4 Protocol driver instance.

Prototype

typedef
    EFI_STATUS
    (EFIAPI *EFI_IP4_CONFIGURE) (
        IN EFI_IP4_PROTOCOL *This,
        IN EFI_IP4_CONFIG_DATA *IpConfigData OPTIONAL
    );

Parameters
This Pointer to the EFI_IP4_PROTOCOL instance.
IpConfigData Pointer to the EFI IPv4 Protocol configuration data structure. Type EFI_IP4_CONFIG_DATA is defined in EFI_IP4_PROTOCOL.GetModeData().

Description
The Configure() function is used to set, change, or reset the operational parameters and filter settings for this EFI IPv4 Protocol instance. Until these parameters have been set, no network traffic can be sent or received by this instance. Once the parameters have been reset (by calling this function with IpConfigData set to NULL), no more traffic can be sent or received until these parameters have been set again. Each EFI IPv4 Protocol instance can be started and stopped independently of each other by enabling or disabling their receive filter settings with the Configure() function.

When IpConfigData.UseDefaultAddress is set to FALSE, the new station address will be appended as an alias address into the addresses list in the EFI IPv4 Protocol driver. While set to TRUE, Configure() will trigger the EFI_IP4_CONFIG2_PROTOCOL to retrieve the default IPv4 address if it is not available yet. Clients could frequently call GetModeData() to check the status to ensure that the default IPv4 address is ready.

If operational parameters are reset or changed, any pending transmit and receive requests will be cancelled. Their completion token status will be set to EFI_ABORTED and their events will be signaled.

Status Codes Returned

<table>
<thead>
<tr>
<th>Code</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>EFI_SUCCESS</td>
<td>The driver instance was successfully opened.</td>
</tr>
<tr>
<td>EFI_NO_MAPPING</td>
<td>When using the default address, configuration (DHCP, BOOTP, RARP, etc.) is not finished yet.</td>
</tr>
<tr>
<td>EFI_IP_ADDRESS_CONFLICT</td>
<td>There is an address conflict in response to the Arp invocation</td>
</tr>
<tr>
<td>EFI_INVALID_PARAMETER</td>
<td>One or more of the following conditions is TRUE:</td>
</tr>
<tr>
<td></td>
<td>- This is NULL</td>
</tr>
<tr>
<td></td>
<td>- IpConfigData.StationAddress is not a unicast IPv4 address.</td>
</tr>
<tr>
<td></td>
<td>- IpConfigData.SubnetMask is not a valid IPv4 subnet mask.</td>
</tr>
</tbody>
</table>
EFI_IP4_PROTOCOL.Groups()

Summary
Joins and leaves multicast groups.

Prototype

typedef
EFI_STATUS
(EIFIAPI *EFI_IP4_GROUPS) (  
    IN EFI_IP4_PROTOCOL *This,
    IN BOOLEAN          JoinFlag,
    IN EFI_IPv4_ADDRESS *GroupAddress OPTIONAL);

Parameters

This Pointer to the EFI_IP4_PROTOCOL instance.

JoinFlag Set to TRUE to join the multicast group session and FALSE to leave.

GroupAddress Pointer to the IPv4 multicast address.

Description
The Groups() function is used to join and leave multicast group sessions. Joining a group will enable reception of matching multicast packets. Leaving a group will disable the multicast packet reception.

If JoinFlag is FALSE and GroupAddress is NULL, all joined groups will be left.
Status Codes Returned

<table>
<thead>
<tr>
<th>Status Code</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>EFI_SUCCESS</td>
<td>The operation completed successfully.</td>
</tr>
<tr>
<td>EFI_INVALID_PARAMETER</td>
<td>One or more of the following is TRUE:</td>
</tr>
<tr>
<td></td>
<td>• This is NULL.</td>
</tr>
<tr>
<td></td>
<td>• JoinFlag is TRUE and GroupAddress is NULL</td>
</tr>
<tr>
<td></td>
<td>• GroupAddress is not NULL and * GroupAddress is not a multicast IPv4 address.</td>
</tr>
<tr>
<td>EFI_NOT_STARTED</td>
<td>This instance has not been started.</td>
</tr>
<tr>
<td>EFI_NO_MAPPING</td>
<td>When using the default address, configuration (DHCP, BOOTP, RARP, etc.) is not finished yet.</td>
</tr>
<tr>
<td>EFI_OUT_OF_RESOURCES</td>
<td>System resources could not be allocated.</td>
</tr>
<tr>
<td>EFI_UNSUPPORTED</td>
<td>This EFI IPv4 Protocol implementation does not support multicast groups.</td>
</tr>
<tr>
<td>EFI_ALREADY_STARTED</td>
<td>The group address is already in the group table (when JoinFlag is TRUE).</td>
</tr>
<tr>
<td>EFI_NOT_FOUND</td>
<td>The group address is not in the group table (when JoinFlag is FALSE).</td>
</tr>
<tr>
<td>EFI_DEVICE_ERROR</td>
<td>An unexpected system or network error occurred.</td>
</tr>
</tbody>
</table>

**EFI_IP4_PROTOCOL.Routes()**

**Summary**
Adds and deletes routing table entries.

**Prototype**

```
typedef
EFI_STATUS
(EIFIAPI *EFI_IP4_ROUTES) (  
    IN EFI_IP4_PROTOCOL *This,
    IN BOOLEAN DeleteRoute,
    IN EFI_IPv4_ADDRESS *SubnetAddress,
    IN EFI_IPv4_ADDRESS *SubnetMask,
    IN EFI_IPv4_ADDRESS *GatewayAddress
    );
```

**Parameters**

- **This** Pointer to the EFI_IP4_PROTOCOL instance.
- **DeleteRoute** Set to TRUE to delete this route from the routing table. Set to FALSE to add this route to the routing table. SubnetAddress and SubnetMask are used as the key to each route entry.
- **SubnetAddress** The address of the subnet that needs to be routed.
- **SubnetMask** The subnet mask of SubnetAddress.
- **GatewayAddress** The unicast gateway IPv4 address for this route.
**Description**

The `Routes()` function adds a route to or deletes a route from the routing table.

Routes are determined by comparing the `SubnetAddress` with the destination IPv4 address arithmetically AND-ed with the `SubnetMask`. The gateway address must be on the same subnet as the configured station address.

The default route is added with `SubnetAddress` and `SubnetMask` both set to 0.0.0.0. The default route matches all destination IPv4 addresses that do not match any other routes.

A `GatewayAddress` that is zero is a nonroute. Packets are sent to the destination IP address if it can be found in the ARP cache or on the local subnet. One automatic nonroute entry will be inserted into the routing table for outgoing packets that are addressed to a local subnet (gateway address of 0.0.0.0).

Each EFI IPv4 Protocol instance has its own independent routing table. Those EFI IPv4 Protocol instances that use the default IPv4 address will also have copies of the routing table that was provided by the `EFI_IP4_CONFIG2_PROTOCOL`, and these copies will be updated whenever the EIF IPv4 Protocol driver reconfigures its instances. As a result, client modification to the routing table will be lost.

**Note:** There is no way to set up routes to other network interface cards because each network interface card has its own independent network stack that shares information only through `EFI IPv4 variable`.

**Status Codes Returned**

<table>
<thead>
<tr>
<th>Code</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>EFI_SUCCESS</td>
<td>The operation completed successfully.</td>
</tr>
<tr>
<td>EFI_NOT_STARTED</td>
<td>The driver instance has not been started.</td>
</tr>
<tr>
<td>EFI_NO_MAPPING</td>
<td>When using the default address, configuration (DHCP, BOOTP, RARP, etc.) is not finished yet.</td>
</tr>
<tr>
<td>EFI_INVALID_PARAMETER</td>
<td>One or more of the following conditions is <strong>true</strong>:</td>
</tr>
<tr>
<td></td>
<td>• This is <strong>NULL</strong>.</td>
</tr>
<tr>
<td></td>
<td>• SubnetAddress is <strong>NULL</strong>.</td>
</tr>
<tr>
<td></td>
<td>• SubnetMask is <strong>NULL</strong>.</td>
</tr>
<tr>
<td></td>
<td>• GatewayAddress is <strong>NULL</strong>.</td>
</tr>
<tr>
<td></td>
<td>• *SubnetAddress is not a valid subnet address.</td>
</tr>
<tr>
<td></td>
<td>• *SubnetMask is not a valid subnet mask.</td>
</tr>
<tr>
<td></td>
<td>• *GatewayAddress is not a valid unicast IPv4 address.</td>
</tr>
<tr>
<td>EFI_OUT_OF_RESOURCES</td>
<td>Could not add the entry to the routing table.</td>
</tr>
<tr>
<td>EFI_NOT_FOUND</td>
<td>This route is not in the routing table (when <code>DeleteRoute</code> is <strong>true</strong>).</td>
</tr>
<tr>
<td>EFI_ACCESS_DENIED</td>
<td>The route is already defined in the routing table (when <code>DeleteRoute</code> is <strong>false</strong>).</td>
</tr>
</tbody>
</table>

**EFI_IP4_PROTOCOL.Transmit()**

**Summary**

Places outgoing data packets into the transmit queue.
Prototype

typedef
    EFI_STATUS
    (EFIAPI *EFI_IP4_TRANSMIT) (  
        IN EFI_IP4_PROTOCOL *This,  
        IN EFI_IP4_COMPLETION_TOKEN *Token  
    );

Parameters

This
	Pointer to the EFI_IP4_PROTOCOL instance.

Token
	Pointer to the transmit token. Type
	EFI_IP4_COMPLETION_TOKEN is defined in “Related Definitions” below.

Description

The Transmit() function places a sending request in the transmit queue of this EFI IPv4 Protocol instance. Whenever the packet in the token is sent out or some errors occur, the event in the token will be signaled and the status is updated.

Related Definitions

    //**********************************************
    // EFI_IP4_COMPLETION_TOKEN
    //**********************************************
    typedef struct {
        EFI_EVENT Event;
        EFI_STATUS Status;
        union {
            EFI_IP4_RECEIVE_DATA *RxData;
            EFI_IP4_TRANSMIT_DATA *TxData;
        }
        Packet;
    } EFI_IP4_COMPLETION_TOKEN;

    Event
	This Event will be signaled after the Status field is updated by the EFI IPv4 Protocol driver. The type of Event must be EFI_NOTIFY_SIGNAL. The Task Priority Level (TPL) of Event must be lower than or equal to TPL_CALLBACK.

    Status
	Will be set to one of the following values:

	EFI_SUCCESS. The receive or transmit completed successfully.

	EFI_ABORTED. The receive or transmit was aborted.

	EFI_TIMEOUT. The transmit timeout expired.

	EFI_ICMP_ERROR. An ICMP error packet was received.

	EFI_DEVICE_ERROR. An unexpected system or network error occurred.

	EFI_NO_MEDIA. There was a media error
**RxData**

When this token is used for receiving, `RxData` is a pointer to the `EFI_IP4_RECEIVE_DATA`. Type `EFI_IP4_RECEIVE_DATA` is defined below.

**TxData**

When this token is used for transmitting, `TxData` is a pointer to the `EFI_IP4_TRANSMIT_DATA`. Type `EFI_IP4_TRANSMIT_DATA` is defined below.

**EFI_IP4_COMPLETION_TOKEN** structures are used for both transmit and receive operations.

When the structure is used for transmitting, the `Event` and `TxData` fields must be filled in by the EFI IPv4 Protocol client. After the transmit operation completes, EFI IPv4 Protocol updates the `Status` field and the `Event` is signaled.

When the structure is used for receiving, only the `Event` field must be filled in by the EFI IPv4 Protocol client. After a packet is received, the EFI IPv4 Protocol fills in the `RxData` and `Status` fields and the `Event` is signaled. If the packet is an ICMP error message, the `Status` is set to `EFI_ICMP_ERROR`, and the packet is delivered up as usual. The protocol from the IP head in the ICMP error message is used to de-multiplex the packet.

```c
//**********************************************
// EFI_IP4_RECEIVE_DATA
//**********************************************
typedef struct {
    EFI_TIME      TimeStamp;
    EFI_EVENT     RecycleSignal;
    UINT32        HeaderLength;
    EFI_IP4_HEADER *Header;
    UINT32        OptionsLength;
    VOID          *Options;
    UINT32        DataLength;
    UINT32        FragmentCount;
    EFI_IP4_FRAGMENT_DATA FragmentTable[1];
} EFI_IP4_RECEIVE_DATA;
```

**TimeStamp**

Time when the EFI IPv4 Protocol driver accepted the packet. `TimeStamp` is zero filled if receive timestamps are disabled or unsupported.

**RecycleSignal**

After this event is signaled, the receive data structure is released and must not be referenced.

**HeaderLength**

Length of the IPv4 packet header. Zero if `ConfigData.RawData` is `TRUE`.

**Header**

Pointer to the IPv4 packet header. If the IPv4 packet was fragmented, this argument is a pointer to the header in the first fragment. `NULL` if `ConfigData.RawData` is `TRUE`. Type `EFI_IP4_HEADER` is defined below.

**OptionsLength**

Length of the IPv4 packet header options. May be zero.
Options  Pointer to the IPv4 packet header options. If the IPv4 packet was fragmented, this argument is a pointer to the options in the first fragment. May be **NULL**.

DataLength  Sum of the lengths of IPv4 packet buffers in FragmentTable. May be zero.

FragmentCount  Number of IPv4 payload (or raw) fragments. If ConfigData.RawData is **TRUE**, this count is the number of raw IPv4 fragments received so far. May be zero.

FragmentTable  Array of payload (or raw) fragment lengths and buffer pointers. If ConfigData.RawData is **TRUE**, each buffer points to a raw IPv4 fragment and thus IPv4 header and options are included in each buffer. Otherwise, IPv4 headers and options are not included in these buffers. Type **EFI_IP4_FRAGMENT_DATA** is defined below.

The EFI IPv4 Protocol receive data structure is filled in when IPv4 packets have been assembled (or when raw packets have been received). In the case of IPv4 packet assembly, the individual packet fragments are only verified and are not reorganized into a single linear buffer.

The FragmentTable contains a sorted list of zero or more packet fragment descriptors. The referenced packet fragments may not be in contiguous memory locations.

```
//****************************************************************************
// EFI_IP4_HEADER
//****************************************************************************
#pragma pack(1)
typedef struct {
    UINT8     HeaderLength:4;
    UINT8     Version:4;
    UINT8     TypeOfService;
    UINT16    TotalLength;
    UINT16    Identification;
    UINT16    Fragmentation;
    UINT8     TimeToLive;
    UINT8     Protocol;
    UINT16    Checksum;
    EFI_IPv4_ADDRESS     SourceAddress;
    EFI_IPv4_ADDRESS     DestinationAddress;
} EFI_IP4_HEADER;
#pragma pack()```

The fields in the IPv4 header structure are defined in the Internet Protocol version 4 specification, which can be found at “Links to UEFI-Related Documents” ([http://uefi.org/uefi](http://uefi.org/uefi)) under the heading “Internet Protocol version 4 Specification”.
The `EFI_IP4_FRAGMENT_DATA` structure describes the location and length of the IPv4 packet fragment to transmit or that has been received.

```c
typedef struct {
    UINT32     FragmentLength;
    VOID       *FragmentBuffer;
} EFI_IP4_FRAGMENT_DATA;
```

- **FragmentLength**: Length of fragment data. This field may not be set to zero.
- **FragmentBuffer**: Pointer to fragment data. This field may not be set to `NULL`.

The `EFI_IP4_TRANSMIT_DATA` structure describes a possibly fragmented packet to be transmitted.

```c
typedef struct {
    EFI_IPv4_ADDRESS       DestinationAddress;
    EFI_IP4.OverrideData   *OverrideData;
    UINT32                 OptionsLength;
    VOID                    *OptionsBuffer;
    UINT32                 TotalDataLength;
    UINT32                 FragmentCount;
    EFI_IP4_FRAGMENT_DATA   FragmentTable[1];
} EFI_IP4_TRANSMIT_DATA;
```

- **DestinationAddress**: The destination IPv4 address. Ignored if `RawData` is `TRUE`.
- **OverrideData**: If not `NULL`, the IPv4 transmission control override data. Ignored if `RawData` is `TRUE`. Type `EFI_IP4_OVERRIDE_DATA` is defined below.
- **OptionsLength**: Length of the IPv4 header options data. Must be zero if the IPv4 driver does not support IPv4 options. Ignored if `RawData` is `TRUE`.
- **OptionsBuffer**: Pointer to the IPv4 header options data. Ignored if `OptionsLength` is zero. Ignored if `RawData` is `TRUE`.
- **TotalDataLength**: Total length of the `FragmentTable` data to transmit.
- **FragmentCount**: Number of entries in the fragment data table.
- **FragmentTable**: Start of the fragment data table. Type `EFI_IP4_FRAGMENT_DATA` is defined above.
typedef struct {
  EFI_IPv4_ADDRESS SourceAddress;
  EFI_IPv4_ADDRESS GatewayAddress;
  UINT8 Protocol;
  UINT8 TypeOfService;
  UINT8 TimeToLive;
  BOOLEAN DoNotFragment;
} EFI_IP4_OVERRIDE_DATA;

SourceAddress Source address override.
GatewayAddress Gateway address to override the one selected from the routing table. This address must be on the same subnet as this station address. If set to 0.0.0.0, the gateway address selected from routing table will not be overridden.
Protocol Protocol type override.
TypeOfService Type-of-service override.
TimeToLive Time-to-live override.
DoNotFragment Do-not-fragment override.

The information and flags in the override data structure will override default parameters or settings for one Transmit() function call.

**Status Codes Returned**

<table>
<thead>
<tr>
<th>Code</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>EFI_SUCCESS</td>
<td>The data has been queued for transmission.</td>
</tr>
<tr>
<td>EFI_NOT_STARTED</td>
<td>This instance has not been started.</td>
</tr>
<tr>
<td>EFI_NO_MAPPING</td>
<td>When using the default address, configuration (DHCP, BOOTP, RARP, etc.) is not finished yet.</td>
</tr>
<tr>
<td>Error Code</td>
<td>Description</td>
</tr>
<tr>
<td>-----------------------</td>
<td>---------------------------------------------------------------------------------------------------------------------------------------------</td>
</tr>
</tbody>
</table>
| EFI_INVALID_PARAMETER | One or more of the following is **TRUE**:  
• This is **NULL**.  
• Token is **NULL**.  
• Token.Event is **NULL**.  
• Token.Packet.TxData is **NULL**.  
• Token.Packet.TxData.OverrideData.GatewayAddress in the override data structure is not a unicast IPv4 address if OverrideData is not **NULL**.  
• Token.Packet.TxData.OverrideData.SourceAddress is not a unicast IPv4 address if OverrideData is not **NULL**.  
• Token.Packet.OptionsLength is not zero and Token.Packet.OptionsBuffer is **NULL**.  
• Token.Packet.ForceCount is zero.  
• One or more of the Token.Packet.TxData.FragmentTable[].FragmentLength fields is zero.  
• One or more of the Token.Packet.TxData.FragmentTable[].FragmentBuffer fields is **NULL**.  
• Token.Packet.TxData.TotalDataLength is zero or not equal to the sum of fragment lengths.  
• The IP header in FragmentTable is not a well-formed header when RawData is **TRUE**. |
| EFI_ACCESS_DENIED     | The transmit completion token with the same Token.Event was already in the transmit queue.                                                    |
| EFI_NOT_READY         | The completion token could not be queued because the transmit queue is full.                                                                  |
| EFI_NOT_FOUND         | Not route is found to destination address.                                                                                                    |
| EFI_OUT_OF_RESOURCES  | Could not queue the transmit data.                                                                                                             |
| EFI_BUFFER_TOO_SMALL  | **Token.Packet.TxData.TotalDataLength** is too short to transmit.                                                                             |
| EFI_BAD_BUFFER_SIZE   | The length of the IPv4 header + option length + total data length is greater than MTU (or greater than the maximum packet size if Token.Packet.TxData.OverrideData.DoNotFragment is **TRUE**.) |
| EFI_NO_MEDIA          | There was a media error.                                                                                                                      |

**EFI_IP4_PROTOCOL.Receive()**

**Summary**

Places a receiving request into the receiving queue.
Prototype

```c
typedef
EFI_STATUS
(EFIAPI *EFI_IP4_RECEIVE) (
    IN EFI_IP4_PROTOCOL *This,
    IN EFI_IP4_COMPLETION_TOKEN *Token
);
```

Parameters

- **This**: Pointer to the `EFI_IP4_PROTOCOL` instance.
- **Token**: Pointer to a token that is associated with the receive data descriptor. Type `EFI_IP4_COMPLETION_TOKEN` is defined in “Related Definitions” of above `Transmit()`.

Description

The `Receive()` function places a completion token into the receive packet queue. This function is always asynchronous.

The `Token.Event` field in the completion token must be filled in by the caller and cannot be `NULL`. When the receive operation completes, the EFI IPv4 Protocol driver updates the `Token.Status` and `Token.Packet.RxData` fields and the `Token.Event` is signaled.

Status Codes Returned

<table>
<thead>
<tr>
<th>Code</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>EFI_SUCCESS</td>
<td>The receive completion token was cached.</td>
</tr>
<tr>
<td>EFI_NOT_STARTED</td>
<td>This EFI IPv4 Protocol instance has not been started.</td>
</tr>
<tr>
<td>EFI_NO_MAPPING</td>
<td>When using the default address, configuration (DHCP, BOOTP, RARP, etc.) is not finished yet.</td>
</tr>
<tr>
<td>EFI_INVALID_PARAMETER</td>
<td>One or more of the following conditions is <strong>TRUE</strong>:</td>
</tr>
<tr>
<td></td>
<td>- <strong>This</strong> is <code>NULL</code>.</td>
</tr>
<tr>
<td></td>
<td>- <strong>Token</strong> is <code>NULL</code>.</td>
</tr>
<tr>
<td></td>
<td>- <strong>Token.Event</strong> is <code>NULL</code>.</td>
</tr>
<tr>
<td>EFI_OUT_OF_RESOURCES</td>
<td>The receive completion token could not be queued due to a lack of system resources (usually memory).</td>
</tr>
<tr>
<td>EFI_DEVICE_ERROR</td>
<td>An unexpected system or network error occurred.</td>
</tr>
<tr>
<td></td>
<td>- The EFI IPv4 Protocol instance has been reset to startup defaults.</td>
</tr>
<tr>
<td>EFI_ACCESS_DENIED</td>
<td>The receive completion token with the same <code>Token.Event</code> was already in the receive queue.</td>
</tr>
<tr>
<td>EFI_NOT_READY</td>
<td>The receive request could not be queued because the receive queue is full.</td>
</tr>
<tr>
<td>EFI_ICMP_ERROR</td>
<td>An ICMP error packet was received.</td>
</tr>
<tr>
<td>EFI_NO_MEDIA</td>
<td>There was a media error.</td>
</tr>
</tbody>
</table>
EFI_IP4_PROTOCOL.Cancel()

Summary
Abort an asynchronous transmit or receive request.

Prototype

typedef
EFI_STATUS
(EFI_API *EFI_IP4_CANCEL)(
    IN EFI_IP4_PROTOCOL *This,
    IN EFI_IP4_COMPLETION_TOKEN *Token OPTIONAL
);

Parameters
This Pointer to the EFI_IP4_PROTOCOL instance.
Token Pointer to a token that has been issued by
        EFI_IP4_PROTOCOL.Transmit() or
        EFI_IP4_PROTOCOL.Receive(). If NULL, all pending tokens are
        aborted. Type EFI_IP4_COMPLETION_TOKEN is defined in
        EFI_IP4_PROTOCOL.Transmit().

Description
The Cancel() function is used to abort a pending transmit or receive request. If the token is in the
transmit or receive request queues, after calling this function, Token->Status will be set to
EFI_ABORTED and then Token->Event will be signaled. If the token is not in one of the queues, which
usually means the asynchronous operation has completed, this function will not signal the token and
EFI_NOT_FOUND is returned.

Status Codes Returned

<table>
<thead>
<tr>
<th>Code</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>EFI_SUCCESS</td>
<td>The asynchronous I/O request was aborted and Token.-&gt;Event was signaled. When Token is NULL, all pending requests were aborted and their events were signaled.</td>
</tr>
<tr>
<td>EFI_INVALID_PARAMETER</td>
<td>This is NULL.</td>
</tr>
<tr>
<td>EFI_NOT_STARTED</td>
<td>This instance has not been started.</td>
</tr>
<tr>
<td>EFI_NO_MAPPING</td>
<td>When using the default address, configuration (DHCP, BOOTP, RARP, etc.) is not finished yet.</td>
</tr>
<tr>
<td>EFI_NOT_FOUND</td>
<td>When Token is not NULL, the asynchronous I/O request was not found in the transmit or receive queue. It has either completed or was not issued by Transmit() and Receive().</td>
</tr>
</tbody>
</table>

EFI_IP4_PROTOCOL.Poll()

Summary
Polls for incoming data packets and processes outgoing data packets.
Prototype

```c
typedef EFI_STATUS
  (EFIAPI *EFI_IP4_POLL) (
    IN EFI_IP4_PROTOCOL *This
  );
```

Parameters

- **This**: Pointer to the `EFI_IP4_PROTOCOL` instance.

Description

The `Poll()` function polls for incoming data packets and processes outgoing data packets. Network drivers and applications can call the `EFI_IP4_PROTOCOL.Poll()` function to increase the rate that data packets are moved between the communications device and the transmit and receive queues.

In some systems the periodic timer event may not poll the underlying communications device fast enough to transmit and/or receive all data packets without missing incoming packets or dropping outgoing packets. Drivers and applications that are experiencing packet loss should try calling the `EFI_IP4_PROTOCOL.Poll()` function more often.

Status Codes Returned

<table>
<thead>
<tr>
<th>Code</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>EFI_SUCCESS</td>
<td>Incoming or outgoing data was processed.</td>
</tr>
<tr>
<td>EFI_NOT_STARTED</td>
<td>This EFI IPv4 Protocol instance has not been started.</td>
</tr>
<tr>
<td>EFI_INVALID_PARAMETER</td>
<td><code>This</code> is <code>NULL</code>.</td>
</tr>
<tr>
<td>EFI_DEVICE_ERROR</td>
<td>An unexpected system or network error occurred.</td>
</tr>
<tr>
<td>EFI_NOT_READY</td>
<td>No incoming or outgoing data is processed.</td>
</tr>
<tr>
<td>EFI_TIMEOUT</td>
<td>Data was dropped out of the transmit and/or receive queue. Consider increasing the polling rate.</td>
</tr>
</tbody>
</table>

28.4 EFI IPv4 Configuration Protocol

This section provides a detailed description of the EFI IPv4 Configuration Protocol.

**IMPORTANT NOTICE**: The `EFI_IP4_CONFIG_PROTOCOL` has been replaced with the new `EFI_IP4_CONFIG2_PROTOCOL`.

- All new designs based on this specification should exclusively use `EFI_IP4_CONFIG2_PROTOCOL`.
- The `EFI_IP4_CONFIG_PROTOCOL` will be removed in the next revision of this specification.

**EFI_IP4_CONFIG_PROTOCOL**

**IMPORTANT NOTICE**: The `EFI_IP4_CONFIG_PROTOCOL` has been replaced with the new `EFI_IP4_CONFIG2_PROTOCOL`.

- All new designs based on this specification should exclusively use `EFI_IP4_CONFIG2_PROTOCOL`.
- The `EFI_IP4_CONFIG_PROTOCOL` will be removed in the next revision of this specification.
Summary
The **EFI_IP4_CONFIG_PROTOCOL** driver performs platform- and policy-dependent configuration for the EFI IPv4 Protocol driver.

GUID
```
#define EFI_IP4_CONFIG_PROTOCOL_GUID \
{0x3b95aa31,0x3793,0x434b,\ 
{0x86,0x67,0xc8,0x07,0x08,0x92,0xe0,0x5e}}
```

Protocol Interface Structure
```
typedef struct _EFI_IP4_CONFIG_PROTOCOL {
  EFI_IP4_CONFIG_START   Start;
  EFI_IP4_CONFIG_STOP    Stop;
  EFI_IP4_CONFIG_GET_DATA GetData;
} EFI_IP4_CONFIG_PROTOCOL;
```

Parameters
- **Start** Starts running the configuration policy for the EFI IPv4 Protocol driver. See the `Start()` function description.
- **Stop** Stops running the configuration policy for the EFI IPv4 Protocol driver. See the `Stop()` function description.
- **GetData** Returns the default configuration data (if any) for the EFI IPv4 Protocol driver. See the `GetData()` function description.

Description
In an effort to keep platform policy code out of the EFI IPv4 Protocol driver, the **EFI_IP4_CONFIG_PROTOCOL** driver will be used as the central repository of any platform- and policy-specific configuration for the EFI IPv4 Protocol driver.

An EFI IPv4 Configuration Protocol interface will be installed on each communications device handle that is managed by the platform setup policy. The driver that is responsible for creating EFI IPv4 variable must open the EFI IPv4 Configuration Protocol driver interface **BY_DRIVER|EXCLUSIVE**.

An example of a configuration policy decision for the EFI IPv4 Protocol driver would be to use a static IP address/subnet mask pair on the platform management network interface and then use dynamic IP addresses that are configured by DHCP on the remaining network interfaces.

**EFI_IP4_CONFIG_PROTOCOL.Start()**

**IMPORTANT NOTICE:** The **EFI_IP4_CONFIG_PROTOCOL** has been replaced with the new **EFI_IP4_CONFIG2_PROTOCOL**.

- All new designs based on this specification should exclusively use **EFI_IP4_CONFIG2_PROTOCOL**.
- The **EFI_IP4_CONFIG_PROTOCOL** will be removed in the next revision of this specification.

Summary
Starts running the configuration policy for the EFI IPv4 Protocol driver.
Prototype

```c
typedef EFI_STATUS (EFI_API *EFI_IP4_CONFIG_START) (
    IN EFI_IP4_CONFIG_PROTOCOL *This,
    IN EFI_EVENT DoneEvent,
    IN EFI_EVENT ReconfigEvent
    );
```

Parameters

- **This**: Pointer to the **EFI_IP4_CONFIG_PROTOCOL** instance.
- **DoneEvent**: Event that will be signaled when the EFI IPv4 Protocol driver configuration policy completes execution. This event must be of type **EVT_NOTIFY_SIGNAL**.
- **ReconfigEvent**: Event that will be signaled when the EFI IPv4 Protocol driver configuration needs to be updated. This event must be of type **EVT_NOTIFY_SIGNAL**.

Description

The **Start()** function is called to determine and to begin the platform configuration policy by the EFI IPv4 Protocol driver. This determination may be as simple as returning **EFI_UNSUPPORTED** if there is no EFI IPv4 Protocol driver configuration policy. It may be as involved as loading some defaults from nonvolatile storage, downloading dynamic data from a DHCP server, and checking permissions with a site policy server.

Starting the configuration policy is just the beginning. It may finish almost instantly or it may take several minutes before it fails to retrieve configuration information from one or more servers. Once the policy is started, drivers should use the **DoneEvent** parameter to determine when the configuration policy has completed. **EFI_IP4_CONFIG_PROTOCOL.GetData()** must then be called to determine if the configuration succeeded or failed.

Until the configuration completes successfully, EFI IPv4 Protocol driver instances that are attempting to use default configurations must return **EFI_NO_MAPPING**.

Once the configuration is complete, the EFI IPv4 Configuration Protocol driver signals **DoneEvent**. The configuration may need to be updated in the future, however; in this case, the EFI IPv4 Configuration Protocol driver must signal **ReconfigEvent**, and all EFI IPv4 Protocol driver instances that are using default configurations must return **EFI_NO_MAPPING** until the configuration policy has been rerun.
Status Codes Returned

<table>
<thead>
<tr>
<th>Status Code</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>EFI_SUCCESS</td>
<td>The configuration policy for the EFI IPv4 Protocol driver is now running.</td>
</tr>
<tr>
<td>EFI_INVALID_PARAMETER</td>
<td>One or more of the following parameters is <strong>NULL</strong>:</td>
</tr>
<tr>
<td></td>
<td>• <strong>This</strong></td>
</tr>
<tr>
<td></td>
<td>• <strong>DoneEvent</strong></td>
</tr>
<tr>
<td></td>
<td>• <strong>ReconfigEvent</strong></td>
</tr>
<tr>
<td>EFI_OUT_OF_RESOURCES</td>
<td>Required system resources could not be allocated.</td>
</tr>
<tr>
<td>EFI_ALREADY_STARTED</td>
<td>The configuration policy for the EFI IPv4 Protocol driver was already started.</td>
</tr>
<tr>
<td>EFI_DEVICE_ERROR</td>
<td>An unexpected system error or network error occurred.</td>
</tr>
<tr>
<td>EFI_UNSUPPORTED</td>
<td>This interface does not support the EFI IPv4 Protocol driver configuration.</td>
</tr>
</tbody>
</table>

**EFI_IP4_CONFIG_PROTOCOL.Stop()**

**IMPORTANT NOTICE:** The **EFI_IP4_CONFIG_PROTOCOL** has been replaced with the new **EFI_IP4_CONFIG2_PROTOCOL**.

- All new designs based on this specification should exclusively use **EFI_IP4_CONFIG2_PROTOCOL**.
- The **EFI_IP4_CONFIG_PROTOCOL** will be removed in the next revision of this specification.

**Summary**

Stops running the configuration policy for the EFI IPv4 Protocol driver.

**Prototype**

```c
typedef EFI_STATUS (EFIAPI *EFI_IP4_CONFIG_STOP) (
    IN EFI_IP4_CONFIG_PROTOCOL *This
);
```

**Parameters**

- **This** Pointer to the **EFI_IP4_CONFIG_PROTOCOL** instance.

**Description**

The **Stop()** function stops the configuration policy for the EFI IPv4 Protocol driver. All configuration data will be lost after calling **Stop()**.
Status Codes Returned

<table>
<thead>
<tr>
<th>Code</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>EFI_SUCCESS</td>
<td>The configuration policy for the EFI IPv4 Protocol driver has been stopped.</td>
</tr>
<tr>
<td>EFI_INVALID_PARAMETER</td>
<td>This is <strong>NULL</strong>.</td>
</tr>
<tr>
<td>EFI_NOT_STARTED</td>
<td>The configuration policy for the EFI IPv4 Protocol driver was not started.</td>
</tr>
</tbody>
</table>

**EFI_IP4_CONFIG_PROTOCOL.GetData()**

**IMPORTANT NOTICE:** The **EFI_IP4_CONFIG_PROTOCOL** has been replaced with the new **EFI_IP4_CONFIG2_PROTOCOL**.

- All new designs based on this specification should exclusively use **EFI_IP4_CONFIG2_PROTOCOL**.
- The **EFI_IP4_CONFIG_PROTOCOL** will be removed in the next revision of this specification.

**Summary**

Returns the default configuration data (if any) for the EFI IPv4 Protocol driver.

**Prototype**

```c
typedef EFI_STATUS
    (EFIAPIC *EFI_IP4_CONFIG_GET_DATA) (
        IN EFI_IP4_CONFIG_PROTOCOL *This,
        IN OUT UINTN *IpConfigDataSize,
        OUT EFI_IP4_IPCONFIG_DATA *IpConfigData OPTIONAL
    );
```

**Parameters**

- **This**
  - Pointer to the **EFI_IP4_CONFIG_PROTOCOL** instance.
- **IpConfigDataSize**
  - On input, the size of the IpConfigData buffer. On output, the count of bytes that were written into the IpConfigData buffer.
- **IpConfigData**
  - Pointer to the EFI IPv4 Configuration Protocol driver configuration data structure. Type **EFI_IP4_IPCONFIG_DATA** is defined in “Related Definitions” below.

**Description**

The **GetData()** function returns the current configuration data for the EFI IPv4 Protocol driver after the configuration policy has completed.
Related Definitions

```c
typedef struct {
    EFI_IPv4_ADDRESS   StationAddress;
    EFI_IPv4_ADDRESS   SubnetMask;
    UINT32             RouteTableSize;
    EFI_IP4_ROUTE_TABLE *RouteTable  OPTIONAL;
} EFI_IP4_IPCONFIG_DATA;
```

- **StationAddress**: Default station IP address, stored in network byte order.
- **SubnetMask**: Default subnet mask, stored in network byte order.
- **RouteTableSize**: Number of entries in the following `RouteTable`. May be zero.
- **RouteTable**: Default routing table data (stored in network byte order). Ignored if `RouteTableSize` is zero. Type `EFI_IP4_ROUTE_TABLE` is defined in `EFI_IP4_PROTOCOL.GetModeData()`.

**EFI_IP4_IPCONFIG_DATA** contains the minimum IPv4 configuration data that is needed to start basic network communication. The `StationAddress` and `SubnetMask` must be a valid unicast IP address and subnet mask.

If `RouteTableSize` is not zero, then `RouteTable` contains a properly formatted routing table for the `StationAddress/SubnetMask`, with the last entry in the table being the default route.

### Status Codes Returned

<table>
<thead>
<tr>
<th>Status Code</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>EFI_SUCCESS</td>
<td>The EFI IPv4 Protocol driver configuration has been returned.</td>
</tr>
<tr>
<td>EFI_INVALID_PARAMETER</td>
<td>This is <strong>NULL</strong>.</td>
</tr>
<tr>
<td>EFI_NOT_STARTED</td>
<td>The configuration policy for the EFI IPv4 Protocol driver is not running.</td>
</tr>
<tr>
<td>EFI_NOT_READY</td>
<td>EFI IPv4 Protocol driver configuration is still running.</td>
</tr>
<tr>
<td>EFI_ABORTED</td>
<td>EFI IPv4 Protocol driver configuration could not complete.</td>
</tr>
<tr>
<td>EFI_BUFFER_TOO_SMALL</td>
<td><em>IpConfigDataSize</em> is smaller than the configuration data buffer or <em>IpConfigData</em> is <strong>NULL</strong>.</td>
</tr>
</tbody>
</table>

### 28.5 EFI IPv4 Configuration II Protocol

This section provides a detailed description of the EFI IPv4 Configuration II Protocol.

**EFI_IP4_CONFIG2_PROTOCOL**

#### Summary

The **EFI_IP4_CONFIG2_PROTOCOL** provides the mechanism to set and get various types of configurations for the EFI IPv4 network stack.
GUID

#define EFI_IP4_CONFIG2_PROTOCOL_GUID \
  { 0x5b446ed1, 0xe30b, 0x4faa, \ 
  { 0x87, 0x1a, 0x36, 0x54, 0xec, 0xa3, 0x60, 0x80 }}

Protocol Interface Structure

typedef struct _EFI_IP4_CONFIG2_PROTOCOL {
  EFI_IP4_CONFIG2_SET_DATA SetData;
  EFI_IP4_CONFIG2_GET_DATA GetData;
  EFI_IP4_CONFIG2_REGISTER_NOTIFY RegisterDataNotify;
  EFI_IP4_CONFIG2_UNREGISTER_NOTIFY UnregisterDataNotify;
} EFI_IP4_CONFIG2_PROTOCOL;

Parameters

SetData
Set the configuration for the EFI IPv4 network stack running on the communication device this EFI IPv4 Configuration II Protocol instance manages. See the SetData() function description.

GetData
Get the configuration for the EFI IPv4 network stack running on the communication device this EFI IPv4 Configuration II Protocol instance manages. See the GetData() function description.

RegisterDataNotify
Register an event that is to be signaled whenever a configuration process on the specified configuration data is done.

UnregisterDataNotify
Remove a previously registered event for the specified configuration data.

Description

The EFI_IP4_CONFIG2_PROTOCOL is designed to be the central repository for the common configurations and the administrator configurable settings for the EFI IPv4 network stack.

An EFI IPv4 Configuration II Protocol instance will be installed on each communication device that the EFI IPv4 network stack runs on.

Note: All the network addresses described in EFI_IP4_CONFIG2_PROTOCOL are stored in network byte order. All other parameters defined in functions or data structures are stored in host byte order.

EFI_IP4_CONFIG2_PROTOCOL.SetData()

Summary
Set the configuration for the EFI IPv4 network stack running on the communication device this EFI IPv4 Configuration II Protocol instance manages.
Prototype

```c
typedef EFI_STATUS
    (EFIAPI *EFI_IP4_CONFIG2_SET_DATA) (  
    IN EFI_IP4_CONFIG2_PROTOCOL  *This,
    IN EFI_IP4_CONFIG2_DATA_TYPE  DataType,
    IN UINTN                     DataSize,
    IN VOID                      *Data
    );
```

Parameters

- **This**
  - Pointer to the `EFI_IP4_CONFIG2_PROTOCOL` instance.
- **DataType**
  - The type of data to set. Type `EFI_IP4_CONFIG2_DATA_TYPE` is defined in “Related Definitions” below.
- **DataSize**
  - Size of the buffer pointed to by `Data` in bytes.
- **Data**
  - The data buffer to set. The type of the data buffer is associated with the `DataType`. The various types are defined in “Related Definitions” below.

Description

This function is used to set the configuration data of type `DataType` for the EFI IPv4 network stack running on the communication device this EFI IPv4 Configuration II Protocol instance manages. The successfully configured data is valid after system reset or power-off.

The `DataSize` is used to calculate the count of structure instances in the `Data` for some `DataType` that multiple structure instances are allowed.

This function is always non-blocking. When setting some type of configuration data, an asynchronous process is invoked to check the correctness of the data, such as doing address conflict detection on the manually set local IPv4 address. `EFI_NOT_READY` is returned immediately to indicate that such an asynchronous process is invoked and the process is not finished yet. The caller willing to get the result of the asynchronous process is required to call `RegisterDataNotify()` to register an event on the specified configuration data. Once the event is signaled, the caller can call `GetData()` to get back the configuration data in order to know the result. For other types of configuration data that do not require an asynchronous configuration process, the result of the operation is immediately returned.
Related Definitions

//***********************************************************
// EFI_IP4_CONFIG2_DATA_TYPE
//***********************************************************
typedef enum {
    Ip4Config2DataTypeInterfaceInfo,
    Ip4Config2DataTypePolicy,
    Ip4Config2DataTypeManualAddress,
    Ip4Config2DataTypeGateway,
    Ip4Config2DataTypeDnsServer,
    Ip4Config2DataTypeMaximum
} EFI_IP4_CONFIG2_DATA_TYPE;

Ip4Config2DataTypeInterfaceInfo
The interface information of the communication device this EFI IPv4 Configuration II Protocol instance manages. This type of data is read only. The corresponding Data is of type EFI_IP4_CONFIG2_INTERFACE_INFO.

Ip4Config2DataTypePolicy
The general configuration policy for the EFI IPv4 network stack running on the communication device this EFI IPv4 Configuration II Protocol instance manages. The policy will affect other configuration settings. The corresponding Data is of type EFI_IP4_CONFIG2_POLICY.

Ip4Config2DataTypeManualAddress
The station addresses set manually for the EFI IPv4 network stack. It is only configurable when the policy is Ip4Config2PolicyStatic. The corresponding Data is of type EFI_IP4_CONFIG2_MANUAL_ADDRESS. When DataSize is 0 and Data is NULL, the existing configuration is cleared from the EFI IPv4 Configuration II Protocol instance.

Ip4Config2DataTypeGateway
The gateway addresses set manually for the EFI IPv4 network stack running on the communication device this EFI IPv4 Configuration II Protocol manages. It is not configurable when the policy is Ip4Config2PolicyDhcp. The gateway addresses must be unicast IPv4 addresses. The corresponding Data is a pointer to an array of EFI_IPv4_ADDRESS instances. When DataSize is 0 and Data is NULL, the existing configuration is cleared from the EFI IPv4 Configuration II Protocol instance.
Ip4Config2DataTypeDnsServer

The DNS server list for the EFI IPv4 network stack running on the communication device this EFI IPv4 Configuration II Protocol manages. It is not configurable when the policy is Ip4Config2PolicyDhcp. The DNS server addresses must be unicast IPv4 addresses. The corresponding Data is a pointer to an array of EFI_IPv4_ADDRESS instances. When DataSize is 0 and Data is NULL, the existing configuration is cleared from the EFI IPv4 Configuration II Protocol instance.

//***********************************************************
// EFI_IP4_CONFIG2_INTERFACE_INFO related definitions
//***********************************************************
#define EFI_IP4_CONFIG2_INTERFACE_INFO_NAME_SIZE 32//

EFI_IP4_CONFIG2_INTERFACE_INFO
//***********************************************************
typedef struct {
  CHAR16     Name[EFI_IP4_CONFIG2_INTERFACE_INFO_NAME_SIZE];
  UINT8      IfType;
  UINT32     HwAddressSize;
  EFI_MAC_ADDRESS  HwAddress;
  EFI_IPv4_ADDRESS StationAddress;
  EFI_IPv4_ADDRESS SubnetMask;
  UINT32     RouteTableSize;
  EFI_IP4_ROUTE_TABLE  *RouteTable OPTIONAL;
} EFI_IP4_CONFIG2_INTERFACE_INFO;

Name     The name of the interface. It is a NULL-terminated Unicode string.
IfType    The interface type of the network interface. See RFC 1700, section “Number Hardware Type”.
HwAddressSize The size, in bytes, of the network interface's hardware address.
HwAddress  The hardware address for the network interface.
StationAddress  The station IPv4 address of this EFI IPv4 network stack.
SubnetMask   The subnet address mask that is associated with the station address.
RouteTableSize Size of the following RouteTable, in bytes. May be zero.
RouteTable   The route table of the IPv4 network stack runs on this interface. Set to NULL if RouteTableSize is zero. Type EFI_IP4_ROUTE_TABLE is defined in EFI_IP4_PROTOCOL.GetModeData().
The **EFI_IP4_CONFIG2_INTERFACE_INFO** structure describes the operational state of the interface this EFI IPv4 Configuration II Protocol instance manages. This type of data is read-only. When reading, the caller allocated buffer is used to return all of the data, i.e., the first part of the buffer is **EFI_IP4_CONFIG2_INTERFACE_INFO** and the followings are the route table if present. The caller should NOT free the buffer pointed to by **RouteTable**, and the caller is only required to free the whole buffer if the data is not needed any more.

```c
typedef enum {
    Ip4Config2PolicyStatic,
    Ip4Config2PolicyDhcp,
    Ip4Config2PolicyMax
} EFI_IP4_CONFIG2_POLICY;
```

**Ip4Config2PolicyStatic**

Under this policy, the **Ip4Config2DataTypeManualAddress**, **Ip4Config2DataTypeGateway** and **Ip4Config2DataTypeDnsServer** configuration data are required to be set manually. The EFI IPv4 Protocol will get all required configuration such as IPv4 address, subnet mask and gateway settings from the EFI IPv4 Configuration II protocol.

**Ip4Config2PolicyDhcp**

Under this policy, the **Ip4Config2DataTypeManualAddress**, **Ip4Config2DataTypeGateway** and **Ip4Config2DataTypeDnsServer** configuration data are not allowed to set via **SetData()**. All of these configurations are retrieved from DHCP server or other auto-configuration mechanism.

The **EFI_IP4_CONFIG2_POLICY** defines the general configuration policy the EFI IPv4 Configuration II Protocol supports. The default policy for a newly detected communication device is beyond the scope of this document. An implementation might leave it to platform to choose the default policy.

The configuration data of type **Ip4Config2DataTypeManualAddress**, **Ip4Config2DataTypeGateway** and **Ip4Config2DataTypeDnsServer** will be flushed if the policy is changed from **Ip4Config2PolicyStatic** to **Ip4Config2PolicyDhcp**.
```c
typedef struct {
  EFI_IPv4_ADDRESS Address;
  EFI_IPv4_ADDRESS SubnetMask;
} EFI_IP4_CONFIG2_MANUAL_ADDRESS;
```

**Address**
The IPv4 unicast address.

**SubnetMask**
The subnet mask.

The `EFI_IP4_CONFIG2_MANUAL_ADDRESS` structure is used to set the station address information for the EFI IPv4 network stack manually when the policy is `Ip4Config2PolicyStatic`.

The `EFI_IP4_CONFIG2_DATA_TYPE` includes current supported data types; this specification allows future extension to support more data types.

### Status Codes Returned

<table>
<thead>
<tr>
<th>Code</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>EFI_SUCCESS</td>
<td>The specified configuration data for the EFI IPv4 network stack is set successfully.</td>
</tr>
<tr>
<td>EFI_INVALID_PARAMETER</td>
<td>One or more of the following are <strong>TRUE</strong>:</td>
</tr>
<tr>
<td></td>
<td>• This is <strong>NULL</strong>.</td>
</tr>
<tr>
<td></td>
<td>• One or more fields in <code>Data</code> and <code>DataSize</code> do not match the requirement of the data type indicated by <code>DataType</code>.</td>
</tr>
<tr>
<td>EFI_WRITE_PROTECTED</td>
<td>The specified configuration data is read-only or the specified configuration data can not be set under the current policy.</td>
</tr>
<tr>
<td>EFI_ACCESS_DENIED</td>
<td>Another set operation on the specified configuration data is already in process.</td>
</tr>
<tr>
<td>EFI_NOT_READY</td>
<td>An asynchronous process is invoked to set the specified configuration data and the process is not finished yet.</td>
</tr>
<tr>
<td>EFI_BAD_BUFFER_SIZE</td>
<td>The <code>DataSize</code> does not match the size of the type indicated by <code>DataType</code>.</td>
</tr>
<tr>
<td>EFI_UNSUPPORTED</td>
<td>This <code>DataType</code> is not supported.</td>
</tr>
<tr>
<td>EFI_OUT_OF_RESOURCES</td>
<td>Required system resources could not be allocated.</td>
</tr>
<tr>
<td>EFI_DEVICE_ERROR</td>
<td>An unexpected system error or network error occurred.</td>
</tr>
</tbody>
</table>

**EFI_IP4_CONFIG2_PROTOCOL.GetData()**

**Summary**

Get the configuration data for the EFI IPv4 network stack running on the communication device this EFI IPv4 Configuration II Protocol instance manages.
Prototype

typedef EFI_STATUS (EFIAPI *EFI_IP4_CONFIG2_GET_DATA) (IN EFI_IP4_CONFIG2_PROTOCOL *This,
IN EFI_IP4_CONFIG2_DATA_TYPE DataType,
IN OUT UINTN *DataSize,
IN VOID *Data OPTIONAL);

Parameters

This Pointer to the EFI_IP4_CONFIG2_PROTOCOL instance.
DataType The type of data to get. Type EFI_IP4_CONFIG2_DATA_TYPE is defined in EFI_IP4_CONFIG2_PROTOCOL.SetData().
DataSize On input, in bytes, the size of Data. On output, in bytes, the size of buffer required to store the specified configuration data.
Data The data buffer in which the configuration data is returned. The type of the data buffer is associated with the DataType. Ignored if DataSize is 0. The various types are defined in EFI_IP4_CONFIG2_PROTOCOL.SetData().

Description

This function returns the configuration data of type DataType for the EFI IPv4 network stack running on the communication device this EFI IPv4 Configuration II Protocol instance manages.

The caller is responsible for allocating the buffer used to return the specified configuration data and the required size will be returned to the caller if the size of the buffer is too small.

EFI_NOT_READY is returned if the specified configuration data is not ready due to an already in progress asynchronous configuration process. The caller can call RegisterDataNotify() to register an event on the specified configuration data. Once the asynchronous configuration process is finished, the event will be signaled and a subsequent GetData() call will return the specified configuration data.

Status Codes Returned

<table>
<thead>
<tr>
<th>Status Code</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>EFI_SUCCESS</td>
<td>The specified configuration data is got successfully.</td>
</tr>
<tr>
<td>EFI_INVALID_PARAMETER</td>
<td>One or more of the followings are TRUE:</td>
</tr>
<tr>
<td></td>
<td>• This is NULL.</td>
</tr>
<tr>
<td></td>
<td>• DataSize is NULL.</td>
</tr>
<tr>
<td></td>
<td>• Data is NULL if *DataSize is not zero.</td>
</tr>
<tr>
<td>EFI_BUFFER_TOO_SMALL</td>
<td>The size of Data is too small for the specified configuration data and the</td>
</tr>
<tr>
<td></td>
<td>required size is returned in DataSize.</td>
</tr>
<tr>
<td>EFI_NOT_READY</td>
<td>The specified configuration data is not ready due to an already in progress</td>
</tr>
<tr>
<td></td>
<td>asynchronous configuration process.</td>
</tr>
<tr>
<td>EFI_NOT_FOUND</td>
<td>The specified configuration data is not found.</td>
</tr>
</tbody>
</table>
EFI_IP4_CONFIG2_PROTOCOL.RegisterDataNotify ()

Summary
Register an event that is to be signaled whenever a configuration process on the specified configuration data is done.

Prototype
typedef
EFI_STATUS
(EIFIAPIF *EFI_IP4_CONFIG2_REGISTER_NOTIFY) (
    IN EFI_IP4_CONFIG2_PROTOCOL *This,
    IN EFI_IP4_CONFIG2_DATA_TYPE DataType,
    IN EFI_EVENT Event
);

Parameters
This Pointer to the EFI_IP4_CONFIG2_PROTOCOL instance.
DataType The type of data to unregister the event for. Type EFI_IP4_CONFIG2_DATA_TYPE is defined in EFI_IP4_CONFIG2_PROTOCOL.SetData().
Event The event to register.

Description
This function registers an event that is to be signaled whenever a configuration process on the specified configuration data is done. An event can be registered for different DataType simultaneously and the caller is responsible for determining which type of configuration data causes the signaling of the event in such case.

Status Codes Returned

<table>
<thead>
<tr>
<th>Status Code</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>EFI_SUCCESS</td>
<td>The notification event for the specified configuration data is registered.</td>
</tr>
<tr>
<td>EFI_INVALID_PARAMETER</td>
<td>This is NULL or Event is NULL</td>
</tr>
<tr>
<td>EFI_UNSUPPORTED</td>
<td>The configuration data type specified by DataType is not supported.</td>
</tr>
<tr>
<td>EFI_OUT_OF_RESOURCES</td>
<td>Required system resources could not be allocated.</td>
</tr>
<tr>
<td>EFI_ACCESS_DENIED</td>
<td>The Event is already registered for the DataType.</td>
</tr>
</tbody>
</table>

EFI_IP4_CONFIG2_PROTOCOL.UnregisterDataNotify ()

Summary
Remove a previously registered event for the specified configuration data.
Prototype

```c
typedef
EFI_STATUS
(EFIAPI *EFI_IP4_CONFIG2_UNREGISTER_NOTIFY) (
    IN EFI_IP4_CONFIG2_PROTOCOL *This,
    IN EFI_IP4_CONFIG2_DATA_TYPE DataType,
    IN EFI_EVENT Event
);
```

Parameters

- **This** - Pointer to the `EFI_IP4_CONFIG2_PROTOCOL` instance.
- **DataType** - The type of data to remove the previously registered event for. Type `EFI_IP4_CONFIG2_DATA_TYPE` is defined in `EFI_IP4_CONFIG2_PROTOCOL.SetData()`.
- **Event** - The event to unregister.

Description

This function removes a previously registered event for the specified configuration data.

Status Codes Returned

<table>
<thead>
<tr>
<th>Status Code</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>EFI_SUCCESS</td>
<td>The event registered for the specified configuration data is removed.</td>
</tr>
<tr>
<td>EFI_INVALID_PARAMETER</td>
<td>This is NULL or Event is NULL.</td>
</tr>
<tr>
<td>EFI_NOT_FOUND</td>
<td>The Event has not been registered for the specified DataType.</td>
</tr>
</tbody>
</table>

### 28.6 EFI IPv6 Protocol

This section defines the EFI IPv6 (Internet Protocol version 6) Protocol interface. It is split into the following three main sections:

- EFI IPv6 Service Binding Protocol
- EFI IPv6 Variable
- EFI IPv6 Protocol

The EFI IPv6 Protocol provides basic network IPv6 packet I/O services, which includes support for Neighbor Discovery Protocol (ND), Multicast Listener Discovery Protocol (MLD), and a subset of the Internet Control Message Protocol (ICMPv6).

#### 28.6.1 IPv6 Service Binding Protocol

**EFI_IP6_SERVICE_BINDING_PROTOCOL**

**Summary**

The EFI IPv6 Service Binding Protocol is used to locate communication devices that are supported by an EFI IPv6 Protocol driver and to create and destroy EFI IPv6 Protocol child instances of the IP6 driver that can use the underlying communications device.
GUID
#define EFI_IP6_SERVICE_BINDING_PROTOCOL_GUID _GUID \  
{0xec835dd3,0xfe0f,0x617b,\ 
{0xa6,0x21,0xb3,0x50,0xc3,0xe1,0x33,0x88}}

Description
A network application that requires basic IPv6 I/O services can use one of the protocol handler services, such as BS->LocateHandleBuffer(), to search for devices that publish an EFI IPv6 Service Binding Protocol GUID. Each device with a published EFI IPv6 Service Binding Protocol GUID supports the EFI IPv6 Protocol and may be available for use.

After a successful call to the EFI_IP6_SERVICE_BINDING_PROTOCOL.CreateChild() function, the newly created child EFI IPv6 Protocol driver is in an un-configured state; it is not ready to send and receive data packets.

Before a network application terminates execution, every successful call to the EFI_IP6_SERVICE_BINDING_PROTOCOL.CreateChild() function must be matched with a call to the EFI_IP6_SERVICE_BINDING_PROTOCOL.DestroyChild() function.

28.6.2 IPv6 Protocol

EFI_IP6_PROTOCOL

Summary
The EFI IPv6 Protocol implements a simple packet-oriented interface that can be used by drivers, daemons, and applications to transmit and receive network packets.

GUID
#define EFI_IP6_PROTOCOL_GUID \ 
{0x2c8759d5,0x5c2d,0x66ef,\ 
{0x92,0x5f,0xb6,0x6c,0x10,0x19,0x57,0xe2}}

Protocol Interface Structure
typedef struct _EFI_IP6_PROTOCOL 
{ 
  EFI_IP6_GET_MODE_DATA GetModeData;
  EFI_IP6_CONFIGURE Config;
  EFI_IP6_GROUPS Groups;
  EFI_IP6_ROUTES Routes;
  EFI_IP6_NEIGHBORS Neighbors;
  EFI_IP6_TRANSMIT Transmit;
  EFI_IP6_RECEIVE Receive;
  EFI_IP6_CANCEL Cancel;
  EFI_IP6_POLL Poll;
} EFI_IP6_PROTOCOL;
Parameters

GetModeData  Gets the current operational settings for this instance of the EFI IPv6 Protocol driver. See the GetModeData() function description.

Configure  Changes or resets the operational settings for the EFI IPv6 Protocol. See the Configure() function description.

Groups  Joins and leaves multicast groups. See the Groups() function description.

Routes  Adds and deletes routing table entries. See the Routes() function description.

Neighbors  Adds and deletes neighbor cache entries. See the Neighbors() function description.

Transmit  Places outgoing data packets into the transmit queue. See the Transmit() function description.

Receive  Places a receiving request into the receiving queue. See the Receive() function description.

Cancel  Aborts a pending transmit or receive request. See the Cancel() function description.

Poll  Polls for incoming data packets and processes outgoing data packets. See the Poll() function description.

Description

The EFI_IP6_PROTOCOL defines a set of simple IPv6, and ICMPv6 services that can be used by any network protocol driver, daemon, or application to transmit and receive IPv6 data packets.

Note:  Byte Order: All the IPv6 addresses that are described in EFI_IP6_PROTOCOL are stored in network byte order. Both incoming and outgoing IP packets are also in network byte order. All other parameters that are defined in functions or data structures are stored in host byte order.

EFI_IP6_PROTOCOL.GetModeData()

Summary

Gets the current operational settings for this instance of the EFI IPv6 Protocol driver.

Prototype

typedef
EFI_STATUS
(EFIAPI *EFI_IP6_GET_MODE_DATA) (  
    IN EFI_IP6_PROTOCOL *This,  
    OUT EFI_IP6_MODE_DATA *Ip6ModeData OPTIONAL,  
    OUT EFI_MANAGED_NETWORK_CONFIG_DATA *MnpConfigData OPTIONAL,  
    OUT EFI_SIMPLE_NETWORK_MODE *SnpModeData OPTIONAL  
);
Parameters

- **This**: Pointer to the `EFI_IP6_PROTOCOL` instance.
- **Ip6ModeData**: Pointer to the EFI IPv6 Protocol mode data structure. Type `EFI_IP6_MODE_DATA` is defined in "Related Definitions" below.
- **MnpConfigData**: Pointer to the managed network configuration data structure. Type `EFI_MANAGED_NETWORK_CONFIG_DATA` is defined in `EFI_MANAGED_NETWORK_PROTOCOL.GetModeData()`.
- **SnpData**: Pointer to the simple network mode data structure. Type `EFI_SIMPLE_NETWORK_MODE` is defined in the `EFI_SIMPLE_NETWORK_PROTOCOL`.

Description

The `GetModeData()` function returns the current operational mode data for this driver instance. The data fields in `EFI_IP6_MODE_DATA` are read only. This function is used optionally to retrieve the operational mode data of underlying networks or drivers.

Related Definitions

```
//**********************************************
// EFI_IP6_MODE_DATA
//**********************************************
typedef struct {
  BOOLEAN IsStarted;
  UINT32 MaxPacketSize;
  EFI_IP6_CONFIG_DATA ConfigData;
  BOOLEAN IsConfigured;
  UINT32 AddressCount;
  EFI_IP6_ADDRESS_INFO *AddressList;
  UINT32 GroupCount;
  EFI_IP6_ADDRESS *GroupTable;
  UINT32 RouteCount;
  EFI_IP6_ROUTE_TABLE *RouteTable;
  UINT32 NeighborCount;
  EFI_IP6_NEIGHBOR_CACHE *NeighborCache;
  UINT32 PrefixCount;
  EFI_IP6_ADDRESS_INFO *PrefixTable;
  UINT32 IcmpTypeCount;
  EFI_IP6_ICMP_TYPE *IcmpTypeList;
} EFI_IP6_MODE_DATA;
```

- **IsStarted**: Set to `TRUE` after this EFI IPv6 Protocol instance is started. All other fields in this structure are undefined until this field is `TRUE`. Set to `FALSE` when the EFI IPv6 Protocol instance is stopped.

- **MaxPacketSize**: The maximum packet size, in bytes, of the packet which the upper layer driver could feed.
**ConfigData**

Current configuration settings. Undefined until `IsStarted` is `TRUE`. Type `EFI_IP6_CONFIG_DATA` is defined below.

**IsConfigured**

Set to `TRUE` when the EFI IPv6 Protocol instance is configured. The instance is configured when it has a station address and corresponding prefix length.

Set to `FALSE` when the EFI IPv6 Protocol instance is not configured.

**AddressCount**

Number of configured IPv6 addresses on this interface.

**AddressList**

List of currently configured IPv6 addresses and corresponding prefix lengths assigned to this interface. It is caller's responsibility to free this buffer. Type `EFI_IP6_ADDRESS_INFO` is defined below.

**GroupCount**

Number of joined multicast groups. Undefined until `IsConfigured` is `TRUE`.

**GroupTable**

List of joined multicast group addresses. It is caller's responsibility to free this buffer. Undefined until `IsConfigured` is `TRUE`.

**RouteCount**

Number of entries in the routing table. Undefined until `IsConfigured` is `TRUE`.

**RouteTable**

Routing table entries. It is caller's responsibility to free this buffer. Type `EFI_IP6_ROUTE_TABLE` is defined below.

**NeighborCount**

Number of entries in the neighbor cache. Undefined until `IsConfigured` is `TRUE`.

**NeighborCache**

Neighbor cache entries. It is caller's responsibility to free this buffer. Undefined until `IsConfigured` is `TRUE`. Type `EFI_IP6_NEIGHBOR_CACHE` is defined below.

**PrefixCount**

Number of entries in the prefix table. Undefined until `IsConfigured` is `TRUE`.

**PrefixTable**

On-link Prefix table entries. It is caller's responsibility to free this buffer. Undefined until `IsConfigured` is `TRUE`. Type `EFI_IP6_ADDRESS_INFO` is defined below.

**IcmpTypeCount**

Number of entries in the supported ICMP types list.

**IcmpTypeList**

Array of ICMP types and codes that are supported by this EFI IPv6 Protocol driver. It is caller's responsibility to free this buffer. Type `EFI_IP6_ICMP_TYPE` is defined below.
typedef struct {
    UINT8    DefaultProtocol;
    BOOLEAN  AcceptAnyProtocol;
    BOOLEAN  AcceptIcmpErrors;
    BOOLEAN  AcceptPromiscuous;
    EFI_IPv6_ADDRESS DestinationAddress;
    EFI_IPv6_ADDRESS StationAddress;
    UINT8    TrafficClass;
    UINT8    HopLimit;
    UINT32   FlowLabel;
    UINT32   ReceiveTimeout;
    UINT32   TransmitTimeout;
} EFI_IP6_CONFIG_DATA;

DefaultProtocol For the IPv6 packet to send and receive, this is the default value of the 'Next Header' field in the last IPv6 extension header or in the IPv6 header if there are no extension headers. Ignored when AcceptPromiscuous is TRUE. An updated list of protocol numbers can be found at “Links to UEFI-Related Documents” (http://uefi.org/uefi) under the heading “IANA Assigned Internet Protocol Numbers”. The following values are illegal: 0 (IPv6 Hop-by-Hop Option), 1(ICMP), 2(IGMP), 41(IPv6), 43(Routing Header for IPv6), 44(Fragment Header for IPv6), 59(No Next Header for IPv6), 60(Destination Options for IPv6), 124(ISIS over IPv4).

AcceptAnyProtocol Set to TRUE to receive all IPv6 packets that get through the receive filters. Set to FALSE to receive only the DefaultProtocol IPv6 packets that get through the receive filters. Ignored when AcceptPromiscuous is TRUE.

AcceptIcmpErrors Set to TRUE to receive ICMP error report packets. Ignored when AcceptPromiscuous or AcceptAnyProtocol is TRUE.

AcceptPromiscuous Set to TRUE to receive all IPv6 packets that are sent to any hardware address or any protocol address. Set to FALSE to stop receiving all promiscuous IPv6 packets.

DestinationAddress The destination address of the packets that will be transmitted. Ignored if it is unspecified.

StationAddress The station IPv6 address that will be assigned to this EFI IPv6 Protocol instance. This field can be set and changed only when the EFI IPv6 driver is transitioning from the stopped to the started states. If the StationAddress is specified, the EFI IPv6 Protocol driver will deliver only incoming IPv6 packets
whose destination matches this IPv6 address exactly. The StationAddress is required to be one of currently configured IPv6 addresses. An address containing all zeroes is also accepted as a special case. Under this situation, the IPv6 driver is responsible for binding a source address to this EFI IPv6 protocol instance according to the source address selection algorithm. Only incoming packets destined to the selected address will be delivered to the user. And the selected station address can be retrieved through later GetModeData() call. If no address is available for selecting, EFI_NO_MAPPING will be returned, and the station address will only be successfully bound to this EFI IPv6 protocol instance after IP6ModeData.IsConfigured changed to TRUE.

TrafficClass TrafficClass field in transmitted IPv6 packets. Default value is zero.
HopLimit HopLimit field in transmitted IPv6 packets.
FlowLabel FlowLabel field in transmitted IPv6 packets. Default value is zero.
ReceiveTimeout The timer timeout value (number of microseconds) for the receive timeout event to be associated with each assembled packet. Zero means do not drop assembled packets.
TransmitTimeout The timer timeout value (number of microseconds) for the transmit timeout event to be associated with each outgoing packet. Zero means do not drop outgoing packets.

The EFI_IP6_CONFIG_DATA structure is used to report and change IPv6 session parameters.

```
//******************************************************
// EFI_IP6_ADDRESS_INFO  //******************************************************
typedef struct {
    EFI_IPv6_ADDRESS Address;
    UINT8 PrefixLength;
} EFI_IP6_ADDRESS_INFO;
```

Address The IPv6 address.
PrefixLength The length of the prefix associated with the Address.
//EFI_IP6_ROUTE_TABLE
typedef struct {
    EFI_IPv6_ADDRESS Gateway;
    EFI_IPv6_ADDRESS Destination;
    UINT8 PrefixLength;
} EFI_IP6_ROUTE_TABLE;

Gateway
The IPv6 address of the gateway to be used as the next hop for packets to this prefix. If the IPv6 address is all zeros, then the prefix is on-link.

Destination
The destination prefix to be routed.

PrefixLength
The length of the prefix associated with the Destination.

EFI_IP6_ROUTE_TABLE is the entry structure that is used in routing tables.

//EFI_IP6_NEIGHBOR_CACHE
typedef struct {
    EFI_IPv6_ADDRESS Neighbor;
    EFI_MAC_ADDRESS LinkAddress;
    EFI_IP6_NEIGHBOR_STATE State;
} EFI_IP6_NEIGHBOR_CACHE;

Neighbor
The on-link unicast / anycast IP address of the neighbor.

LinkAddress
Link-layer address of the neighbor.

State
State of this neighbor cache entry.

EFI_IP6_NEIGHBOR_CACHE is the entry structure that is used in neighbor cache. It records a set of entries about individual neighbors to which traffic has been sent recently.
typedef enum {
    EfiNeighborInComplete,
    EfiNeighborReachable,
    EfiNeighborStale,
    EfiNeighborDelay,
    EfiNeighborProbe
} EFI_IP6_NEIGHBOR_STATE;

Following is a description of the fields in the above enumeration.

EfiNeighborInComplete Address resolution is being performed on this entry. Specially, Neighbor Solicitation has been sent to the solicited-node multicast address of the target, but corresponding Neighbor Advertisement has not been received.

EfiNeighborReachable Positive confirmation was received that the forward path to the neighbor was functioning properly.

EfiNeighborStale Reachable Time has elapsed since the last positive confirmation was received. In this state, the forward path to the neighbor was functioning properly.

EfiNeighborDelay This state is an optimization that gives upper-layer protocols additional time to provide reachability confirmation.

EfiNeighborProbe A reachability confirmation is actively sought by retransmitting Neighbor Solicitations every RetransTimer milliseconds until a reachability confirmation is received.

typedef struct {
    UINT8 Type;
    UINT8 Code;
} EFI_IP6_ICMP_TYPE;

The type of ICMP message. See “Links to UEFI-Related Documents” (http://uefi.org/uefi) under the heading “Internet Control Message Protocol Version 6 (ICMPv6) Parameters” for the complete list of ICMP message type.

The code of the ICMP message, which further describes the different ICMP message formats under the same Type. See “Links to UEFI-Related Documents” (http://uefi.org/uefi) under the heading “Internet Control Message Protocol
EFI_IP6_ICMP_TYPE is used to describe those ICMP messages that are supported by this EFI IPv6 Protocol driver.

```c
//***********************************************************
// ICMPv6 type definitions for error messages
//***********************************************************
#define ICMP_V6_DEST_UNREACHABLE 0x1
#define ICMP_V6_PACKET_TOO_BIG 0x2
#define ICMP_V6_TIME_EXCEEDED 0x3
#define ICMP_V6_PARAMETER_PROBLEM 0x4

//***********************************************************
// ICMPv6 type definition for informational messages
//***********************************************************
#define ICMP_V6_ECHO_REQUEST 0x80
#define ICMP_V6_ECHO_REPLY 0x81
#define ICMP_V6_LISTENER_QUERY 0x82
#define ICMP_V6_LISTENER_REPORT 0x83
#define ICMP_V6_LISTENER_DONE 0x84
#define ICMP_V6_ROUTER_SOLICIT 0x85
#define ICMP_V6_ROUTER_ADVERTISE 0x86
#define ICMP_V6_NEIGHBOR_SOLICIT 0x87
#define ICMP_V6_NEIGHBOR_ADVERTISE 0x88
#define ICMP_V6_REDIRECT 0x89
#define ICMP_V6_LISTENER_REPORT_2 0x8F

//***********************************************************
// ICMPv6 code definitions for ICMP_V6_DEST_UNREACHABLE
//***********************************************************
#define ICMP_V6_NO_ROUTE_TO_DEST 0x0
#define ICMP_V6_COMM_PROHIBITED 0x1
#define ICMP_V6_BEYOND_SCOPE 0x2
#define ICMP_V6_ADDR_UNREACHABLE 0x3
#define ICMP_V6_PORT_UNREACHABLE 0x4
#define ICMP_V6_SOURCE_ADDR_FAILED 0x5
#define ICMP_V6_ROUTE_REJECTED 0x6

//***********************************************************
// ICMPv6 code definitions for ICMP_V6_TIME_EXCEEDED
//***********************************************************
#define ICMP_V6_TIMEOUT_HOP_LIMIT 0x0
#define ICMP_V6_TIMEOUT_REASSEMBLE 0x1
```
Status Codes Returned

<table>
<thead>
<tr>
<th>Code</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>EFI_SUCCESS</td>
<td>The operation completed successfully.</td>
</tr>
<tr>
<td>EFI_INVALID_PARAMETER</td>
<td>This is NULL</td>
</tr>
<tr>
<td>EFI_OUT_OF_RESOURCES</td>
<td>The required mode data could not be allocated.</td>
</tr>
</tbody>
</table>

**EFI_IP6_PROTOCOL.Configure()**

**Summary**
Assign IPv6 address and other configuration parameter to this EFI IPv6 Protocol driver instance.

**Prototype**

```c
typedef EFI_STATUS (EFIAPI *EFI_IP6_CONFIGURE) (
    IN EFI_IP6_PROTOCOL *This,
    IN EFI_IP6_CONFIG_DATA *Ip6ConfigData OPTIONAL);
```

**Parameters**

- **This**
  Pointer to the **EFI_IP6_PROTOCOL** instance.
- **Ip6ConfigData**
  Pointer to the EFI IPv6 Protocol configuration data structure.
  Type **EFI_IP6_CONFIG_DATA** is defined in **EFI_IP6_PROTOCOL.GetModeData()**.

**Description**

The **Configure()** function is used to set, change, or reset the operational parameters and filter settings for this EFI IPv6 Protocol instance. Until these parameters have been set, no network traffic can be sent or received by this instance. Once the parameters have been reset (by calling this function with **Ip6ConfigData** set to **NULL**), no more traffic can be sent or received until these parameters have been set again. Each EFI IPv6 Protocol instance can be started and stopped independently of each other by enabling or disabling their receive filter settings with the **Configure()** function.

If **Ip6ConfigData.StationAddress** is a valid non-zero IPv6 unicast address, it is required to be one of the currently configured IPv6 addresses list in the EFI IPv6 drivers, or else **EFI_INVALID_PARAMETER** will be returned. If **Ip6ConfigData.StationAddress** is unspecified, the IPv6 driver will bind a source address according to the source address selection algorithm. Clients could frequently call **GetModeData()** to check get currently configured IPv6 address list in the EFI IPv6 driver. If both
Ip6ConfigData.StationAddress and Ip6ConfigData.Destination are unspecified, when transmitting the packet afterwards, the source address filled in each outgoing IPv6 packet is decided based on the destination of this packet.

If operational parameters are reset or changed, any pending transmit and receive requests will be cancelled. Their completion token status will be set to EFI_ABORTED and their events will be signaled.

**Status Codes Returned**

<table>
<thead>
<tr>
<th>Code</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>EFI_SUCCESS</td>
<td>The driver instance was successfully opened.</td>
</tr>
<tr>
<td>EFI_INVALID_PARAMETER</td>
<td>One or more of the following conditions is TRUE: This is NULL.</td>
</tr>
<tr>
<td></td>
<td>Ip6ConfigData.StationAddress is neither zero nor a unicast IPv6 address.</td>
</tr>
<tr>
<td></td>
<td>Ip6ConfigData.StationAddress is neither zero nor one of the configured IP addresses in the EFI IPv6 driver.</td>
</tr>
<tr>
<td></td>
<td>Ip6ConfigData.DefaultProtocol is illegal.</td>
</tr>
<tr>
<td>EFI_OUT_OF_RESOURCES</td>
<td>The EFI IPv6 Protocol driver instance data could not be allocated.</td>
</tr>
<tr>
<td>EFI_NO_MAPPING</td>
<td>The IPv6 driver was responsible for choosing a source address for this instance, but no source address was available for use.</td>
</tr>
<tr>
<td>EFI_ALREADY_STARTED</td>
<td>The interface is already open and must be stopped before the IPv6 address or prefix length can be changed.</td>
</tr>
<tr>
<td>EFI_DEVICE_ERROR</td>
<td>An unexpected system or network error occurred. The EFI IPv6 Protocol driver instance is not opened.</td>
</tr>
<tr>
<td>EFI_UNSUPPORTED</td>
<td>Default protocol specified through Ip6ConfigData.DefaultProtocol isn’t supported.</td>
</tr>
</tbody>
</table>

**EFI_IP6_PROTOCOL.Groups()**

**Summary**

Joins and leaves multicast groups.

**Prototype**

```c
typedef EFI_STATUS
  (EFIAPI *EFI_IP6_GROUPS) (  
    IN EFI_IP6_PROTOCOL *This,  
    IN BOOLEAN JoinFlag,  
    IN EFI_IPv6_ADDRESS *GroupAddress OPTIONAL  
  );
```

**Parameters**

<table>
<thead>
<tr>
<th>Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>This</td>
<td>Pointer to the EFI_IP6_PROTOCOL instance.</td>
</tr>
<tr>
<td>JoinFlag</td>
<td>Set to TRUE to join the multicast group session and FALSE to leave.</td>
</tr>
<tr>
<td>GroupAddress</td>
<td>Pointer to the IPv6 multicast address.</td>
</tr>
</tbody>
</table>
Description
The Groups() function is used to join and leave multicast group sessions. Joining a group will enable reception of matching multicast packets. Leaving a group will disable reception of matching multicast packets. Source-Specific Multicast isn’t required to be supported.

If JoinFlag is FALSE and GroupAddress is NULL, all joined groups will be left.

Status Codes Returned

<table>
<thead>
<tr>
<th>Status Code</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>EFI_SUCCESS</td>
<td>The operation completed successfully.</td>
</tr>
<tr>
<td>EFI_INVALID_PARAMETER</td>
<td>One or more of the following is TRUE: This is NULL. JoinFlag is TRUE and GroupAddress is NULL. GroupAddress is not NULL and *GroupAddress is not a multicast IPv6 address. GroupAddress is not NULL and *GroupAddress is in the range of SSM destination address.</td>
</tr>
<tr>
<td>EFI_NOT_STARTED</td>
<td>This instance has not been started.</td>
</tr>
<tr>
<td>EFI_OUT_OF_RESOURCES</td>
<td>System resources could not be allocated.</td>
</tr>
<tr>
<td>EFI_UNSUPPORTED</td>
<td>This EFI IPv6 Protocol implementation does not support multicast groups.</td>
</tr>
<tr>
<td>EFI_ALREADY_STARTED</td>
<td>The group address is already in the group table (when JoinFlag is TRUE).</td>
</tr>
<tr>
<td>EFI_NOT_FOUND</td>
<td>The group address is not in the group table (when JoinFlag is FALSE).</td>
</tr>
<tr>
<td>EFI_DEVICE_ERROR</td>
<td>An unexpected system or network error occurred.</td>
</tr>
</tbody>
</table>

EFI_IP6_PROTOCOL.Routes()

Summary
Adds and deletes routing table entries.

Prototype

typedef EFI_STATUS
    (EFIAPI *EFI_IP6_ROUTES) ( 
    IN EFI_IP6_PROTOCOL *This, 
    IN BOOLEAN DeleteRoute, 
    IN EFI_IPv6_ADDRESS *Destination OPTIONAL, 
    IN UINT8 PrefixLength, 
    IN EFI_IPv6_ADDRESS *GatewayAddress OPTIONAL )
;

Parameters
This Pointer to the EFI_IP6_PROTOCOL instance.
DeleteRoute

Set to **TRUE** to delete this route from the routing table. Set to **FALSE** to add this route to the routing table. **Destination**, **PrefixLength** and **Gateway** are used as the key to each route entry.

**Destination**

The address prefix of the subnet that needs to be routed.

**PrefixLength**

The prefix length of **Destination**. Ignored if **Destination** is **NULL**.

**GatewayAddress**

The unicast gateway IPv6 address for this route.

**Description**

The **Routes()** function adds a route to or deletes a route from the routing table.

Routes are determined by comparing the leftmost **PrefixLength** bits of **Destination** with the destination IPv6 address arithmetically. The gateway address must be on the same subnet as the configured station address.

The default route is added with **Destination** and **PrefixLegth** both set to all zeros. The default route matches all destination IPv6 addresses that do not match any other routes.

All EFI IPv6 Protocol instances share a routing table.

**Note:** There is no way to set up routes to other network interface cards because each network interface card has its own independent network stack that shares information only through the EFI IPv6 variable.

**Status Codes Returned**

<table>
<thead>
<tr>
<th>Code</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>EFI_SUCCESS</td>
<td>The operation completed successfully.</td>
</tr>
<tr>
<td>EFI_NOT_STARTED</td>
<td>The driver instance has not been started.</td>
</tr>
<tr>
<td>EFI_INVALID_PARAMETER</td>
<td>One or more of the following conditions is <strong>TRUE</strong>:</td>
</tr>
<tr>
<td></td>
<td>This is <strong>NULL</strong>.</td>
</tr>
<tr>
<td></td>
<td>When <strong>DeleteRoute</strong> is <strong>TRUE</strong>, both <strong>Destination</strong> and <strong>GatewayAddress</strong> are <strong>NULL</strong></td>
</tr>
<tr>
<td></td>
<td>When <strong>DeleteRoute</strong> is <strong>FALSE</strong>, either <strong>Destination</strong> or <strong>GatewayAddress</strong> is <strong>NULL</strong></td>
</tr>
<tr>
<td></td>
<td>+<strong>GatewayAddress</strong> is not a valid unicast IPv6 address.</td>
</tr>
<tr>
<td></td>
<td>+<strong>GatewayAddress</strong> is one of the local configured IPv6 addresses.</td>
</tr>
<tr>
<td>EFI_OUT_OF_RESOURCES</td>
<td>Could not add the entry to the routing table.</td>
</tr>
<tr>
<td>EFI_NOT_FOUND</td>
<td>This route is not in the routing table (when <strong>DeleteRoute</strong> is <strong>TRUE</strong>).</td>
</tr>
<tr>
<td>EFI_ACCESS_DENIED</td>
<td>The route is already defined in the routing table (when <strong>DeleteRoute</strong> is <strong>FALSE</strong>).</td>
</tr>
</tbody>
</table>

**EFI_IP6_PROTOCOL.Neighbors()**

**Summary**

Add or delete Neighbor cache entries.
Prototype

typedef
EFI_STATUS
(EIFIAPI *EFI_IP6_NEIGHBORS) (
   IN EFI_IP6_PROTOCOL     *This,
   IN BOOLEAN              DeleteFlag,
   IN EFI_IPv6_ADDRESS     *TargetIp6Address,
   IN EFI_MAC_ADDRESS      *TargetLinkAddress OPTIONAL
   IN UINT32               Timeout,
   IN BOOLEAN              Override
);

Parameters

This Pointer to the EFI_IP6_PROTOCOL instance.
DeleteFlag Set to TRUE to delete the specified cache entry, set to FALSE to add (or update, if it already exists and Override is TRUE) the specified cache entry. TargetIp6Address is used as the key to find the requested cache entry.
TargetIp6Address Pointer to Target IPv6 address.
TargetLinkAddress Pointer to link-layer address of the target. Ignored if NULL.
Timeout Time in 100-ns units that this entry will remain in the neighbor cache, it will be deleted after Timeout. A value of zero means that the entry is permanent. A non-zero value means that the entry is dynamic.
Override If TRUE, the cached link-layer address of the matching entry will be overridden and updated; if FALSE, EFI_ACCESS_DENIED will be returned if a corresponding cache entry already existed.

Description

The Neighbors() function is used to add, update, or delete an entry from neighbor cache.
IPv6 neighbor cache entries are typically inserted and updated by the network protocol driver as network traffic is processed. Most neighbor cache entries will time out and be deleted if the network traffic stops. Neighbor cache entries that were inserted by Neighbors() may be static (will not timeout) or dynamic (will time out).

The implementation should follow the neighbor cache timeout mechanism which is defined in RFC4861. The default neighbor cache timeout value should be tuned for the expected network environment.
Status Codes Returned

<table>
<thead>
<tr>
<th>Code</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>EFI_SUCCESS</td>
<td>The operation completed successfully.</td>
</tr>
<tr>
<td>EFI_NOT_STARTED</td>
<td>The driver instance has not been started.</td>
</tr>
<tr>
<td>EFI_INVALID_PARAMETER</td>
<td>One or more of the following conditions is TRUE:</td>
</tr>
<tr>
<td></td>
<td>* This is NULL.</td>
</tr>
<tr>
<td></td>
<td>* TargetIpAddress is NULL.</td>
</tr>
<tr>
<td></td>
<td>* TargetLinkAddress is invalid when not NULL.</td>
</tr>
<tr>
<td></td>
<td>* TargetIpAddress is not a valid unicast IPv6 address.</td>
</tr>
<tr>
<td></td>
<td>* TargetIpAddress is one of the local configured IPv6 addresses.</td>
</tr>
<tr>
<td>EFI_OUT_OF_RESOURCES</td>
<td>Could not add the entry to the neighbor cache.</td>
</tr>
<tr>
<td>EFI_NOT_FOUND</td>
<td>This entry is not in the neighbor cache (when DeleteFlag is TRUE or when DeleteFlag is FALSE while TargetLinkAddress is NULL.).</td>
</tr>
<tr>
<td>EFI_ACCESS_DENIED</td>
<td>The to-be-added entry is already defined in the neighbor cache, and that entry is tagged as un-overridden (when DeleteFlag is FALSE).</td>
</tr>
</tbody>
</table>

**EFI_IP6_PROTOCOL.Transmit()**

**Summary**
Places outgoing data packets into the transmit queue.

**Prototype**
```c
typedef EFI_STATUS (EFIAPI *EFI_IP6_TRANSMIT) (  
    IN EFI_IP6_PROTOCOL *This,  
    IN EFI_IP6_COMPLETION_TOKEN *Token  
);
```

**Parameters**
- **This** Pointer to the EFI_IP6_PROTOCOL instance.
- **Token** Pointer to the transmit token. Type EFI_IP6_COMPLETION_TOKEN is defined in "Related Definitions" below.

**Description**
The Transmit() function places a sending request in the transmit queue of this EFI IPv6 Protocol instance. Whenever the packet in the token is sent out or some errors occur, the event in the token will be signaled and the status is updated.
Related Definitions

```c
typedef struct {
    EFI_EVENT Event;
    EFI_STATUS Status;
    union {
        EFI_IP6_RECEIVE_DATA *RxData;
        EFI_IP6_TRANSMIT_DATA *TxData;
    }
} EFI_IP6_COMPLETION_TOKEN;
```

**Event**
This event will be signaled after the Status field is updated by the EFI IPv6 Protocol driver. The type of Event must be `EFI_NOTIFY_SIGNAL`.

**Status**
Will be set to one of the following values:

- **EFI_SUCCESS**: The receive or transmit completed successfully.
- **EFI_ABORTED**: The receive or transmit was aborted.
- **EFI_TIMEOUT**: The transmit timeout expired.
- **EFI_ICMP_ERROR**: An ICMP error packet was received.
- **EFI_DEVICE_ERROR**: An unexpected system or network error occurred.
- **EFI_SECURITY_VIOLATION**: The transmit or receive was failed because of an IPsec policy check.

**RxData**
When the Token is used for receiving, RxData is a pointer to the `EFI_IP6_RECEIVE_DATA`. Type `EFI_IP6_RECEIVE_DATA` is defined below.

**TxData**
When the Token is used for transmitting, TxData is a pointer to the `EFI_IP6_TRANSMIT_DATA`. Type `EFI_IP6_TRANSMIT_DATA` is defined below.

`EFI_IP6_COMPLETION_TOKEN` structures are used for both transmit and receive operations.

When the structure is used for transmitting, the Event and TxData fields must be filled in by the EFI IPv6 Protocol client. After the transmit operation completes, the EFI IPv6 Protocol driver updates the Status field and the Event is signaled.

When the structure is used for receiving, only the Event field must be filled in by the EFI IPv6 Protocol client. After a packet is received, the EFI IPv6 Protocol driver fills in the RxData and Status fields and the Event is signaled.
typedef struct _EFI_IP6_RECEIVE_DATA {
    EFI_TIME TimeStamp;
    EFI_EVENT RecycleSignal;
    UINT32 HeaderLength;
    EFI_IP6_HEADER *Header;
    UINT32 DataLength;
    UINT32 FragmentCount;
    EFI_IP6_FRAGMENT_DATA FragmentTable[1];
} EFI_IP6_RECEIVE_DATA;

TimeStamp
    Time when the EFI IPv6 Protocol driver accepted the packet. TimeStamp is zero filled if timestamps are disabled or unsupported.

RecycleSignal
    After this event is signaled, the receive data structure is released and must not be referenced.

HeaderLength
    Length of the IPv6 packet headers, including both the IPv6 header and any extension headers.

Header
    Pointer to the IPv6 packet header. If the IPv6 packet was fragmented, this argument is a pointer to the header in the first fragment. Type EFI_IP6_HEADER is defined below.

DataLength
    Sum of the lengths of IPv6 packet buffers in FragmentTable. May be zero.

FragmentCount
    Number of IPv6 payload fragments. May be zero.

FragmentTable
    Array of payload fragment lengths and buffer pointers. Type EFI_IP6_FRAGMENT_DATA is defined below.

The EFI IPv6 Protocol receive data structure is filled in when IPv6 packets have been assembled. In the case of IPv6 packet assembly, the individual packet fragments are only verified and are not reorganized into a single linear buffer.

The FragmentTable contains a sorted list of zero or more packet fragment descriptors. The referenced packet fragments may not be in contiguous memory locations.
The fields in the IPv6 header structure are defined in the Internet Protocol version 6 specification, which can be found at “Links to UEFI-Related Documents” (http://uefi.org/uefi) under the heading “Internet Protocol version 6 Specification”.

The **EFI_IP6_FRAGMENT_DATA** structure describes the location and length of the IPv6 packet fragment to transmit or that has been received.
typedef struct _EFI_IP6_TRANSMIT_DATA {
    EFI_IPv6_ADDRESS DestinationAddress;
    EFI_IP6_OVERRIDE_DATA  *OverrideData;
    UINT32 ExtHdrsLength;
    VOID *ExtHdrs;
    UINT8 NextHeader;
    UINT32 DataLength;
    UINT32 FragmentCount;
    EFI_IP6_FRAGMENT_DATA FragmentTable[1];
} EFI_IP6_TRANSMIT_DATA;

DestinationAddress The destination IPv6 address. If it is unspecified, ConfigData.DestinationAddress will be used instead.
OverrideData If not NULL, the IPv6 transmission control override data. Type EFI_IP6_OVERRIDE_DATA is defined below.
ExtHdrsLength Total length in byte of the IPv6 extension headers specified in ExtHdrs
ExtHdrs Pointer to the IPv6 extension headers. The IP layer will append the required extension headers if they are not specified by ExtHdrs. Ignored if ExtHdrsLength is zero.
NextHeader The protocol of first extension header in ExtHdrs. Ignored if ExtHdrsLength is zero.
DataLength Total length in bytes of the FragmentTable data to transmit.
FragmentCount Number of entries in the fragment data table.
FragmentTable Start of the fragment data table. Type EFI_IP6_FRAGMENT_DATA is defined above.

The EFI_IP6_TRANSMIT_DATA structure describes a possibly fragmented packet to be transmitted.

typedef struct _EFI_IP6_OVERRIDE_DATA {
    UINT8 Protocol;
    UINT8 HopLimit;
    UINT32 FlowLabel;
} EFI_IP6_OVERRIDE_DATA;

Protocol Protocol type override.
HopLimit Hop-Limit override.
**FlowLabel**

Flow-Label override.

The information and flags in the override data structure will override default parameters or settings for one `Transmit()` function call.

### Status Codes Returned

<table>
<thead>
<tr>
<th>Code</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>EFI_SUCCESS</td>
<td>The data has been queued for transmission.</td>
</tr>
<tr>
<td>EFI_NOT_STARTED</td>
<td>This instance has not been started.</td>
</tr>
<tr>
<td>EFI_NO_MAPPING</td>
<td>The IPv6 driver was responsible for choosing a source address for this transmission, but no source address was available for use.</td>
</tr>
</tbody>
</table>
| EFI_INVALID_PARAMETER | One or more of the following is **TRUE**:  
  - This is **NULL**.
  - Token is **NULL**.
  - Token.Event is **NULL**.
  - Token.Packet.TxData is **NULL**.
  - Token.Packet.ExtHdrsLength is not zero and Token.Packet.ExtHdrs is **NULL**.
  - Token.Packet.FragmentCount is zero.
  - One or more of the Token.Packet.TxData.FragmentTable[].FragmentLength fields is zero.
  - One or more of the Token.Packet.TxData.FragmentTable[].FragmentBuffer fields is **NULL**.
  - Token.Packet.TxData.DataLength is zero or not equal to the sum of fragment lengths.
  - Token.Packet.TxData.DestinationAddress is non-zero when DestinationAddress is configured as non-zero when doing `Configure()` for this EFI IPv6 protocol instance.
  - Token.Packet.TxData.DestinationAddress is unspecified when DestinationAddress is unspecified when doing `Configure()` for this EFI IPv6 protocol instance. |
| EFI_ACCESS_DENIED | The transmit completion token with the same `Token.Event` was already in the transmit queue. |
| EFI_NOT_READY     | The completion token could not be queued because the transmit queue is full. |
| EFI_NOT_FOUND     | No route was found to destination address.                                |
| EFI_OUT_OF_RESOURCES | Could not queue the transmit data.                                         |
| EFI_BUFFER_TOO_SMALL | `Token.Packet.TxData.DataLength` is too short to transmit.               |
| EFI_BAD_BUFFER_SIZE | If `Token.Packet.TxData.DataLength` is beyond the maximum that which can be described through the Fragment Offset field in Fragment header when performing fragmentation. |
| EFI_DEVICE_ERROR  | An unexpected system or network error occurred.                           |
| EFI_NO_MEDIA      | There was a media error.                                                  |
**EFI_IP6_PROTOCOL.Receive()**

**Summary**
Places a receiving request into the receiving queue.

**Prototype**

```c
typedef EFI_STATUS (EFIAPI *EFI_IP6_RECEIVE) (
    IN EFI_IP6_PROTOCOL *This,
    IN EFI_IP6_COMPLETION_TOKEN *Token
);
```

**Parameters**

- **This** Pointer to the **EFI_IP6_PROTOCOL** instance.
- **Token** Pointer to a token that is associated with the receive data descriptor. Type **EFI_IP6_COMPLETION_TOKEN** is defined in "Related Definitions" of above **Transmit()**.

**Description**

The **Receive()** function places a completion token into the receive packet queue. This function is always asynchronous.

The **Token.Event** field in the completion token must be filled in by the caller and cannot be **NULL**. When the receive operation completes, the EFI IPv6 Protocol driver updates the **Token.Status** and **Token.Packet.RxData** fields and the **Token.Event** is signaled.

**Status Codes Returned**

<table>
<thead>
<tr>
<th>Status Code</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>EFI_SUCCESS</strong></td>
<td>The receive completion token was cached.</td>
</tr>
<tr>
<td><strong>EFI_NOT_STARTED</strong></td>
<td>This EFI IPv6 Protocol instance has not been started.</td>
</tr>
<tr>
<td><strong>EFI_NO_MAPPING</strong></td>
<td>When IP6 driver responsible for binding source address to this instance, while no source address is available for use.</td>
</tr>
<tr>
<td><strong>EFI_INVALID_PARAMETER</strong></td>
<td>One or more of the following conditions is <strong>TRUE</strong>:</td>
</tr>
<tr>
<td></td>
<td><strong>This</strong> is <strong>NULL</strong></td>
</tr>
<tr>
<td></td>
<td><strong>Token</strong> is <strong>NULL</strong></td>
</tr>
<tr>
<td></td>
<td><strong>Token.Event</strong> is <strong>NULL</strong></td>
</tr>
<tr>
<td><strong>EFI_OUT_OF_RESOURCES</strong></td>
<td>The receive completion token could not be queued due to a lack of system resources (usually memory).</td>
</tr>
<tr>
<td><strong>EFI_DEVICE_ERROR</strong></td>
<td>An unexpected system or network error occurred. The EFI IPv6 Protocol instance has been reset to startup defaults.</td>
</tr>
<tr>
<td><strong>EFI_ACCESS_DENIED</strong></td>
<td>The receive completion token with the same <strong>Token.Event</strong> was already in the receive queue.</td>
</tr>
<tr>
<td><strong>EFI_NOT_READY</strong></td>
<td>The receive request could not be queued because the receive queue is full.</td>
</tr>
</tbody>
</table>
EFI_IP6_PROTOCOL.Cancel()

Summary
Abort an asynchronous transmits or receive request.

Prototype

typedef
EFI_STATUS
(EIFIAPIC *EFI_IP6_CANCEL)(
    IN EFI_IP6_PROTOCOL *This,
    IN EFI_IP6_COMPLETION_TOKEN *Token OPTIONAL);

Parameters

This     Pointer to the EFI_IP6_PROTOCOL instance.
Token    Pointer to a token that has been issued by EFI_IP6_PROTOCOL .Transmit() or EFI_IP6_PROTOCOL.Receive(). If NULL, all pending tokens are aborted. Type EFI_IP6_COMPLETION_TOKEN is defined in EFI_IP6_PROTOCOL.Transmit().

Description

The Cancel() function is used to abort a pending transmit or receive request. If the token is in the transmit or receive request queues, after calling this function, Token->Status will be set to EFI_ABORTED and then Token->Event will be signaled. If the token is not in one of the queues, which usually means the asynchronous operation has completed, this function will not signal the token and EFI_NOT_FOUND is returned.

Status Codes Returned

<table>
<thead>
<tr>
<th>Code</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>EFI_SUCCESS</td>
<td>The asynchronous I/O request was aborted and Token-&gt;Event was signaled.</td>
</tr>
<tr>
<td></td>
<td>When Token is NULL, all pending requests were aborted and their events were signaled.</td>
</tr>
<tr>
<td>EFI_INVALID_PARAMETER</td>
<td>This is NULL.</td>
</tr>
<tr>
<td>EFI_NOT_STARTED</td>
<td>This instance has not been started.</td>
</tr>
<tr>
<td>EFI_NOT_FOUND</td>
<td>When Token is not NULL, the asynchronous I/O request was not found in the transmit or receive queue. It has either completed or was not issued by Transmit() and Receive().</td>
</tr>
<tr>
<td>EFI_DEVICE_ERROR</td>
<td>An unexpected system or network error occurred.</td>
</tr>
</tbody>
</table>
**EFI_IP6_PROTOCOL.Poll()**

**Summary**
Polls for incoming data packets and processes outgoing data packets.

**Prototype**
```c
typedef EFI_STATUS (EFIAPI *EFI_IP6_POLL) (IN EFI_IP6_PROTOCOL *This);
```

**Description**
The `Poll()` function polls for incoming data packets and processes outgoing data packets. Network drivers and applications can call the `EFI_IP6_PROTOCOL.Poll()` function to increase the rate that data packets are moved between the communications device and the transmit and receive queues.

In some systems the periodic timer event may not poll the underlying communications device fast enough to transmit and/or receive all data packets without missing incoming packets or dropping outgoing packets. Drivers and applications that are experiencing packet loss should try calling the `EFI_IP6_PROTOCOL.Poll()` function more often.

**Status Codes Returned**

<table>
<thead>
<tr>
<th>Code</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>EFI_SUCCESS</td>
<td>Incoming or outgoing data was processed.</td>
</tr>
<tr>
<td>EFI_NOT_STARTED</td>
<td>This EFI IPv6 Protocol instance has not been started.</td>
</tr>
<tr>
<td>EFI_INVALID_PARAMETER</td>
<td>This is NULL.</td>
</tr>
<tr>
<td>EFI_DEVICE_ERROR</td>
<td>An unexpected system or network error occurred.</td>
</tr>
<tr>
<td>EFI_NOT_READY</td>
<td>No incoming or outgoing data is processed.</td>
</tr>
<tr>
<td>EFI_TIMEOUT</td>
<td>Data was dropped out of the transmit and/or receive queue.</td>
</tr>
<tr>
<td></td>
<td>Consider increasing the polling rate.</td>
</tr>
</tbody>
</table>

**28.7 EFI IPv6 Configuration Protocol**

This section provides a detailed description of the EFI IPv6 Configuration Protocol.

**EFI_IP6_CONFIG_PROTOCOL**

**Summary**
The `EFI_IP6_CONFIG_PROTOCOL` provides the mechanism to set and get various types of configurations for the EFI IPv6 network stack.
GUID

```c
#define EFI_IP6_CONFIG_PROTOCOL_GUID \ 
{0x937fe521,0x95ae,0x4d1a,\ 
{0x89,0x29,0x48,0xbc,0xd9,0x0a,0xd3,0x1a}
```

Protocol Interface Structure

```c
typedef struct _EFI_IP6_CONFIG_PROTOCOL {
    EFI_IP6_CONFIG_SET_DATA     SetData;
    EFI_IP6_CONFIG_GET_DATA     GetData;
    EFI_IP6_CONFIG_REGISTER_NOTIF RegisterDataNotify;
    EFI_IP6_CONFIG_UNREGISTER_NOTIFY UnregisterDataNotify;
} EFI_IP6_CONFIG_PROTOCOL;
```

Parameters

- **SetData**: Set the configuration for the EFI IPv6 network stack running on the communication device this EFI IPv6 Configuration Protocol instance manages. See the `SetData()` function description.

- **GetData**: Get the configuration or register an event to monitor the change of the configuration for the EFI IPv6 network stack running on the communication device this EFI IPv6 Configuration Protocol instance manages. See the `GetData()` function description.

- **RegisterDataNotify**: Register an event that is to be signaled whenever a configuration process on the specified configuration data is done.

- **UnregisterDataNotify**: Remove a previously registered event for the specified configuration data.

Description

The `EFI_IP6_CONFIG_PROTOCOL` is designed to be the central repository for the common configurations and the administrator configurable settings for the EFI IPv6 network stack.

An EFI IPv6 Configuration Protocol instance will be installed on each communication device that the EFI IPv6 network stack runs on.

**EFI_IP6_CONFIG_PROTOCOL.SetData()**

**Summary**

Set the configuration for the EFI IPv6 network stack running on the communication device this EFI IPv6 Configuration Protocol instance manages.
Prototype

```c
typedef EFI_STATUS
    (EFIAPI *EFI_IP6_CONFIG_SET_DATA) ( 
    IN EFI_IP6_CONFIG_PROTOCOL *This,
    IN EFI_IP6_CONFIG_DATA_TYPE DataType,
    IN UINTN DataSize,
    IN VOID *Data
    );
```

Parameters

- **This**: Pointer to the `EFI_IP6_CONFIG_PROTOCOL` instance.
- **DataType**: The type of data to set. Type `EFI_IP6_CONFIG_DATA_TYPE` is defined in "Related Definitions" below.
- **DataSize**: Size of the buffer pointed to by `Data` in bytes.
- **Data**: The data buffer to set. The type of the data buffer is associated with the `DataType`. The various types are defined in "Related Definitions" below.

Description

This function is used to set the configuration data of type `DataType` for the EFI IPv6 network stack running on the communication device this EFI IPv6 Configuration Protocol instance manages.

The `DataSize` is used to calculate the count of structure instances in the `Data` for some `DataType` that multiple structure instances are allowed.

This function is always non-blocking. When setting some type of configuration data, an asynchronous process is invoked to check the correctness of the data, such as doing Duplicate Address Detection on the manually set local IPv6 addresses. `EFI_NOT_READY` is returned immediately to indicate that such an asynchronous process is invoked and the process is not finished yet. The caller willing to get the result of the asynchronous process is required to call `RegisterDataNotify()` to register an event on the specified configuration data. Once the event is signaled, the caller can call `GetData()` to get back the configuration data in order to know the result. For other types of configuration data that do not require an asynchronous configuration process, the result of the operation is immediately returned.
Related Definitions

```
//***********************************************************
// EFI_IP6_CONFIG_DATA_TYPE
//***********************************************************
typedef enum {
  Ip6ConfigDataTypeInterfaceInfo,
  Ip6ConfigDataTypeAltInterfaceId,
  Ip6ConfigDataTypePolicy,
  Ip6ConfigDataTypeDupAddrDetectTransmits,
  Ip6ConfigDataTypeManualAddress,
  Ip6ConfigDataTypeGateway,
  Ip6ConfigDataTypeDnsServer,
  Ip6ConfigDataTypeMaximum
} EFI_IP6_CONFIG_DATA_TYPE;
```

**Ip6ConfigDataTypeInterfaceInfo**
The interface information of the communication device this EFI IPv6 Configuration Protocol instance manages. This type of data is read only. The corresponding Data is of type `EFI_IP6_CONFIG_INTERFACE_INFO`.

**Ip6ConfigDataTypeAltInterfaceId**
The alternative interface ID for the communication device this EFI IPv6 Configuration Protocol instance manages if the link local IPv6 address generated from the interfaced ID based on the default source the EFI IPv6 Protocol uses is a duplicate address. The length of the interface ID is 64 bit. The corresponding Data is of type `EFI_IP6_CONFIG_INTERFACE_ID`.

**Ip6ConfigDataTypePolicy**
The general configuration policy for the EFI IPv6 network stack running on the communication device this EFI IPv6 Configuration Protocol instance manages. The policy will affect other configuration settings. The corresponding Data is of type `EFI_IP6_CONFIG_POLICY`.

**Ip6ConfigDataTypeDupAddrDetectTransmits**
The number of consecutive Neighbor Solicitation messages sent while performing Duplicate Address Detection on a tentative address. A value of zero indicates that Duplicate Address Detection will not be performed on tentative addresses. The corresponding Data is of type `EFI_IP6_CONFIG_DUP_ADDR_DETECT_TRANSMITS`.

**Ip6ConfigDataTypeManualAddress**
The station addresses set manually for the EFI IPv6 network stack. It is only configurable when the policy is `Ip6ConfigPolicyManual`. The corresponding Data is a pointer to an array of `EFI_IPv6_ADDRESS` instances. When DataSize is 0 and Data is `NULL`, the existing configuration is cleared from the EFI IPv6 Configuration Protocol instance.

**Ip6ConfigDataTypeGateway**
The gateway addresses set manually for the EFI IPv6 network stack running on the communication device this EFI IPv6 Configuration Protocol manages. It is not configurable when the policy is `Ip6ConfigPolicyAutomatic`. The gateway
addresses must be unicast IPv6 addresses. The corresponding Data is a pointer to an array of EFI_IPv6_ADDRESS instances. When dataSize is 0 and Data is NULL, the existing configuration is cleared from the EFI IPv6 Configuration Protocol instance.

Ip6ConfigDataTypeDnsServer The DNS server list for the EFI IPv6 network stack running on the communication device this EFI IPv6 Configuration Protocol manages. It is not configurable when the policy is Ip6ConfigPolicyAutomatic. The DNS server addresses must be unicast IPv6 addresses. The corresponding Data is a pointer to an array of EFI_IPv6_ADDRESS instances. When dataSize is 0 and Data is NULL, the existing configuration is cleared from the EFI IPv6 Configuration Protocol instance.

//***********************************************************
// EFI_IP6_CONFIG_INTERFACE_INFO
//***********************************************************
typedef struct {
    CHAR16 Name[32];
    UINT8 IfType;
    UINT32 HwAddressSize;
    EFI_MAC_ADDRESS HwAddress;
    UINT32 AddressInfoCount;
    EFI_IP6_ADDRESS_INFO *AddressInfo;
    UINT32 RouteCount;
    EFI_IP6_ROUTE_TABLE *RouteTable;
} EFI_IP6_CONFIG_INTERFACE_INFO;

Name The name of the interface. It is a NULL-terminated string.
IfType The interface type of the network interface. See RFC 3232, section "Number Hardware Type".
HwAddressSize The size, in bytes, of the network interface's hardware address.
HwAddress The hardware address for the network interface.
AddressInfoCount Number of EFI_IP6_ADDRESS_INFO structures pointed to by AddressInfo.
AddressInfo Pointer to an array of EFI_IP6_ADDRESS_INFO instances which contain the local IPv6 addresses and the corresponding prefix length information. Set to NULL if AddressInfoCount is zero. Type EFI_IP6_ADDRESS_INFO is defined in EFI_IP6_PROTOCOL.GetModeData().
RouteCount Number of route table entries in the following RouteTable.
RouteTable The route table of the IPv6 network stack runs on this interface. Set to NULL if RouteCount is zero. Type
EFI_IP6_ROUTE_TABLE is defined in EFI_IP6_PROTOCOL.GetModeData().

The EFI_IP6_CONFIG_INTERFACE_INFO structure describes the operational state of the interface this EFI IPv6 Configuration Protocol instance manages. This type of data is read-only. When reading, the caller allocated buffer is used to return all of the data, i.e., the first part of the buffer is EFI_IP6_CONFIG_INTERFACE_INFO and the followings are the array of EFI_IP6_ADDRESS_INFO and the route table if present. The caller should NOT free the buffer pointed to by AddressInfo or RouteTable, and the caller is only required to free the whole buffer if the data is not needed any more.

//***************************************************************************
// EFI_IP6_CONFIG_INTERFACE_ID
//***************************************************************************
typedef struct {
    UINT8 Id[8];
} EFI_IP6_CONFIG_INTERFACE_ID;

The EFI_IP6_CONFIG_INTERFACE_ID structure describes the 64-bit interface ID.

//***************************************************************************
// EFI_IP6_CONFIG_POLICY
//***************************************************************************
typedef enum {
    Ip6ConfigPolicyManual,
    Ip6ConfigPolicyAutomatic
} EFI_IP6_CONFIG_POLICY;

Ip6ConfigPolicyManual Under this policy, the
Ip6ConfigDataTypeManualAddress,
Ip6ConfigDataTypeGateway and
Ip6ConfigDataTypeDnsServer configuration data are
required to be set manually. The EFI IPv6 Protocol will get all
required configuration such as address, prefix and gateway
settings from the EFI IPv6 Configuration protocol.

Ip6ConfigPolicyAutomatic Under this policy, the
Ip6ConfigDataTypeManualAddress,
Ip6ConfigDataTypeGateway and
Ip6ConfigDataTypeDnsServer configuration data are not
allowed to set via SetData(). All of these configurations are
retrieved from some auto configuration mechanism. The EFI
IPv6 Protocol will use the IPv6 stateless address
autoconfiguration mechanism and/or the IPv6 stateful
address autoconfiguration mechanism described in the
related RFCs to get address and other configuration
information.

The EFI_IP6_CONFIG_POLICY defines the general configuration policy the EFI IPv6 Configuration
Protocol supports. The default policy for a newly detected communication device is beyond the scope of
this document. An implementation might leave it to platform to choose the default policy.
The configuration data of type `IpI6ConfigDataTypeManualAddress`, `Ip6ConfigDataTypeGateway` and `Ip6ConfigDataTypeDnsServer` will be flushed if the policy is changed from `Ip6ConfigPolicyManual` to `Ip6ConfigPolicyAutomatic`.

```c
typedef struct {  
    UINT32 DupAddrDetectTransmits;
} EFI_IP6_CONFIG_DUP_ADDR_DETECT_TRANSMITS;
```

The `EFI_IP6_CONFIG_DUP_ADDR_DETECT_TRANSMITS` structure describes the number of consecutive Neighbor Solicitation messages sent while performing Duplicate Address Detection on a tentative address. The default value for a newly detected communication device is 1.

```c
typedef struct {  
    EFI_IPv6_ADDRESS Address;
    BOOLEAN IsAnycast;
    UINT8 PrefixLength;
} EFI_IP6_CONFIG_MANUAL_ADDRESS;
```

The `EFI_IP6_CONFIG_MANUAL_ADDRESS` structure is used to set the station address information for the EFI IPv6 network stack manually when the policy is `Ip6ConfigPolicyManual`. Address represents the IPv6 unicast address. IsAnycast is set to `TRUE` if Address is anycast. PrefixLength is the length, in bits, of the prefix associated with this Address.
Status Codes Returned

<table>
<thead>
<tr>
<th>Code</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>EFI_SUCCESS</td>
<td>The specified configuration data for the EFI IPv6 network stack is set successfully.</td>
</tr>
<tr>
<td>EFI_INVALID_PARAMETER</td>
<td>One or more of the following are TRUE:</td>
</tr>
<tr>
<td></td>
<td>- This is NULL.</td>
</tr>
<tr>
<td></td>
<td>- One or more fields in Data and DataSize do not match the requirement of the data type indicated by DataType.</td>
</tr>
<tr>
<td>EFI_WRITE_PROTECTED</td>
<td>The specified configuration data is read-only or the specified configuration data can not be set under the current policy.</td>
</tr>
<tr>
<td>EFI_ACCESS_DENIED</td>
<td>Another set operation on the specified configuration data is already in process.</td>
</tr>
<tr>
<td>EFI_NOT_READY</td>
<td>An asynchronous process is invoked to set the specified configuration data and the process is not finished yet.</td>
</tr>
<tr>
<td>EFI_BAD_BUFFER_SIZE</td>
<td>The DataSize does not match the size of the type indicated by DataType.</td>
</tr>
<tr>
<td>EFI_UNSUPPORTED</td>
<td>This DataType is not supported.</td>
</tr>
<tr>
<td>EFI_OUT_OF_RESOURCES</td>
<td>Required system resources could not be allocated.</td>
</tr>
<tr>
<td>EFI_DEVICE_ERROR</td>
<td>An unexpected system error or network error occurred.</td>
</tr>
</tbody>
</table>

**EFI_IP6_CONFIG_PROTOCOL.GetData()**

**Summary**

Get the configuration data for the EFI IPv6 network stack running on the communication device this EFI IPv6 Configuration Protocol instance manages.

**Prototype**

```c
typedef
  EFI_STATUS
(EFIAPI *EFI_IP6_CONFIG_GET_DATA)(
  IN EFI_IP6_CONFIG_PROTOCOL  *This,
  IN EFI_IP6_CONFIG_DATA_TYPE Data_Type,
  IN OUT UINTN               *DataSize,
  IN VOID                    *Data OPTIONAL
);
```

**Parameters**

<table>
<thead>
<tr>
<th>Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>This</td>
<td>Pointer to the EFI_IP6_CONFIG_PROTOCOL instance.</td>
</tr>
<tr>
<td>DataType</td>
<td>The type of data to get. Type EFI_IP6_CONFIG_DATA_TYPE is defined in EFI_IP6_CONFIG_PROTOCOL.SetData().</td>
</tr>
<tr>
<td>DataSize</td>
<td>On input, in bytes, the size of Data. On output, in bytes, the size of buffer required to store the specified configuration data.</td>
</tr>
</tbody>
</table>
**Data**

The data buffer in which the configuration data is returned. The type of the data buffer is associated with the `DataType`. Ignored if `DataSize` is 0. The various types are defined in `EFI_IP6_CONFIG_PROTOCOL.SetData()`.

**Description**

This function returns the configuration data of type `DataType` for the EFI IPv6 network stack running on the communication device this EFI IPv6 Configuration Protocol instance manages.

The caller is responsible for allocating the buffer used to return the specified configuration data and the required size will be returned to the caller if the size of the buffer is too small.

`EFI_NOT_READY` is returned if the specified configuration data is not ready due to an already in progress asynchronous configuration process. The caller can call ` EFI_IP6_CONFIG_PROTOCOL.RegisterDataNotify()` to register an event on the specified configuration data. Once the asynchronous configuration process is finished, the event will be signaled and a subsequent `GetData()` call will return the specified configuration data.

**Status Codes Returned**

<table>
<thead>
<tr>
<th>Status Code</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>EFI_SUCCESS</td>
<td>The specified configuration data got successfully.</td>
</tr>
<tr>
<td>EFI_INVALID_PARAMETER</td>
<td>One or more of the followings are TRUE:</td>
</tr>
<tr>
<td></td>
<td>• This is <code>NULL</code>.</td>
</tr>
<tr>
<td></td>
<td>• <code>DataSize</code> is <code>NULL</code>.</td>
</tr>
<tr>
<td></td>
<td>• <code>Data</code> is <code>NULL</code> if <code>*DataSize</code> is not zero.</td>
</tr>
<tr>
<td>EFI_BUFFER_TOO_SMALL</td>
<td>The size of <code>Data</code> is too small for the specified configuration data and the required size is returned in <code>DataSize</code>.</td>
</tr>
<tr>
<td>EFI_NOT_READY</td>
<td>The specified configuration data is not ready due to an already in progress asynchronous configuration process.</td>
</tr>
<tr>
<td>EFI_NOT_FOUND</td>
<td>The specified configuration data is not found.</td>
</tr>
</tbody>
</table>

**EFI_IP6_CONFIG_PROTOCOL.RegisterDataNotify()**

**Summary**

Register an event that is to be signaled whenever a configuration process on the specified configuration data is done.
Prototype

typedef
EFI_STATUS
(EIFIAPI *EFI_IP6_CONFIG_REGISTER_NOTIFY) (
    IN EFI_IP6_CONFIG_PROTOCOL  *This,
    IN EFI_IP6_CONFIG_DATA_TYPE  DataType,
    IN EFI_EVENT                Event
);

Parameters

This        Pointer to the EFI_IP6_CONFIG_PROTOCOL instance.
DataType     The type of data to unregister the event for. Type
             EFI_IP6_CONFIG_DATA_TYPE is defined in
             EFI_IP6_CONFIG_PROTOCOL.SetData().
Event        The event to register.

Description

This function registers an event that is to be signaled whenever a configuration process on the specified
configuration data is done. An event can be registered for different DataType simultaneously and the
caller is responsible for determining which type of configuration data causes the signaling of the event in
such case.

Status Codes Returned

<table>
<thead>
<tr>
<th>Status Code</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>EFI_SUCCESS</td>
<td>The notification event for the specified configuration data is registered.</td>
</tr>
<tr>
<td>EFI_INVALID_PARAMETER</td>
<td>This is NULL or Event is NULL.</td>
</tr>
<tr>
<td>EFI_UNSUPPORTED</td>
<td>The configuration data type specified by DataType is not supported.</td>
</tr>
<tr>
<td>EFI_OUT_OF_RESOURCES</td>
<td>Required system resources could not be allocated.</td>
</tr>
<tr>
<td>EFI_ACCESS_DENIED</td>
<td>The Event is already registered for the DataType.</td>
</tr>
</tbody>
</table>

EFI_IP6_CONFIG_PROTOCOL.UnregisterDataNotify ()

Summary

Remove a previously registered event for the specified configuration data.
Prototype

typedef EFI_STATUS
(EFIAPI *EFI_IP6_CONFIG_UNREGISTER_NOTIFY) (
    IN EFI_IP6_CONFIG_PROTOCOL *This,
    IN EFI_IP6_CONFIG_DATA_TYPE DataType,
    IN EFI_EVENT Event
);

Parameters

This
Pointer to the EFI_IP6_CONFIG_PROTOCOL instance.

DataType
The type of data to remove the previously registered event for. Type EFI_IP6_CONFIG_DATA_TYPE is defined in EFI_IP6_CONFIG_PROTOCOL.SetData().

Event
The event to unregister.

Description
This function removes a previously registered event for the specified configuration data.

Status Codes Returned

<table>
<thead>
<tr>
<th>Status Code</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>EFI_SUCCESS</td>
<td>The event registered for the specified configuration data is removed.</td>
</tr>
<tr>
<td>EFI_INVALID_PARAMETER</td>
<td>This is NULL or Event is NULL.</td>
</tr>
<tr>
<td>EFI_NOT_FOUND</td>
<td>The Event has not been registered for the specified DataType.</td>
</tr>
</tbody>
</table>

28.8 IPsec

28.8.1 IPsec Overview

IPsec is a framework of open standards that provides data confidentiality, data integrity, data authentication and replay protection between participating peers. A set of security services is provided by IPsec for traffic at the IP layer, in both the IPv4 and IPv6 environment. To the stronger, IPV6 requires IPsec support.

IPsec is documented in a series of Internet RFCs. The overall IPsec architecture and implementation are guided by “Security Architecture for the Internet Protocol”, RFC 4301.

Two different security protocols – Authentication Header (AH, described in RFC 4302) and Encapsulated Security Payload (ESP, described in RFC 4303) – are used to provide package-level security for IP datagram.

This section attempts to capture the generic configuration for an IPsec implementation in an EFI environment.
28.8.2 EFI IPsec Configuration Protocol

This section provides a detailed description of the EFI IPsec Configuration Protocol. This protocol sets and obtains the IPsec configuration information.

**EFI_IPSEC_CONFIG_PROTOCOL**

**Summary**

The `EFI_IPSEC_CONFIG_PROTOCOL` provides the mechanism to set and retrieve security and policy related information for the EFI IPsec protocol driver.

**GUID**

```c
#define EFI_IPSEC_CONFIG_PROTOCOL_GUID \
{0xce5e5929,0xc7a3,0x4602,\} \
{0xad,0x9e,0xc9,0xda,0xf9,0x4e,0xbf,0xcf}\
```

**Protocol Interface Structure**

```c
typedef struct _EFI_IPSEC_CONFIG_PROTOCOL {
  EFI_IPSEC_CONFIG_SET_DATA    SetData;
  EFI_IPSEC_CONFIG_GET_DATA    GetData;
  EFI_IPSEC_CONFIG_GET_NEXT_SELECTOR GetNextSelector;
  EFI_IPSEC_CONFIG_REGISTER_NOTIFY    RegisterDataNotify;
  EFI_IPSEC_CONFIG_UNREGISTER_NOTIFY    UnregisterDataNotify;
} EFI_IPSEC_CONFIG_PROTOCOL;
```

**Parameters**

- **SetData**: Set the configuration and control information for the EFI IPsec protocol driver. See the `SetData()` function description.
- **GetData**: Look up and retrieve the IPsec configuration data. See the `GetData()` function description.
- **GetNextSelector**: Enumerates the current IPsec configuration data entry selector. See the `GetNextSelector()` function description.
- **RegisterNotify**: Register an event that is to be signaled whenever a configuration process on the specified IPsec configuration data is done.
- **UnregisterNotify**: Remove a registered event for the specified IPsec configuration data.

**Description**

The `EFI_IPSEC_CONFIG_PROTOCOL` provides the ability to set and lookup the IPsec SAD (Security Association Database), SPD (Security Policy Database) data entry and configure the security association management protocol such as IKEv2. This protocol is used as the central repository of any policy-specific configuration for EFI IPsec driver.
EFI_IPSEC_CONFIG_PROTOCOL can be bound to both IPv4 and IPv6 stack. User can use this protocol for IPsec configuration in both IPv4 and IPv6 environment.

EFI_IPSEC_CONFIG_PROTOCOL.SetData()

Summary
Set the security association, security policy and peer authorization configuration information for the EFI IPsec driver.

Prototype

typedef
EFI_STATUS
(EIFIAPI *EFI_IPSEC_CONFIG_SET_DATA) (  
    IN EFI_IPSEC_CONFIG_PROTOCOL    *This,
    IN EFI_IPSEC_CONFIG_DATA_TYPE   DataType,
    IN EFI_IPSEC_CONFIG_SELECTOR    *Selector,
    IN VOID               *Data
    IN EFI_IPSEC_CONFIG_SELECTOR    *InsertBefore OPTIONAL
);

Parameters
This
   Pointer to the EFI_IPSEC_CONFIG_PROTOCOL instance.
InsertBefore
   Pointer to one entry selector which describes the expected position the new data entry will be added. If InsertBefore is NULL, the new entry will be appended the end of database.
DataType
   The type of data to be set. Type EFI_IPSEC_CONFIG_DATA_TYPE is defined in "Related Definitions" below.
Selector
   Pointer to an entry selector on operated configuration data specified by DataType. A NULL Selector causes the entire specified-type configuration information to be flushed.
Data
   The data buffer to be set. The structure of the data buffer is associated with the DataType. The various types are defined in "Related Definitions" below.

Description
This function is used to set the IPsec configuration information of type DataType for the EFI IPsec driver.

The IPsec configuration data has a unique selector/identifier separately to identify a data entry. The selector structure depends on DataType’s definition.

Using SetData() with a Data of NULL causes the IPsec configuration data entry identified by DataType and Selector to be deleted.
Related Definitions

```c
typedef enum {
  IPsecConfigDataTypeSpd,
  IPsecConfigDataTypeSad,
  IPsecConfigDataTypePad,
  IPsecConfigDataTypeMaximum
} EFI_IPSEC_CONFIG_DATA_TYPE;
```

**IPsecConfigDataTypeSpd**

The IPsec Security Policy Database (aka SPD) setting. In IPsec, an essential element of Security Association (SA) processing is the underlying SPD that specifies what services are to be offered to IP datagram and in what fashion. The SPD must be consulted during the processing of all traffic (inbound and outbound), including traffic not protected by IPsec, that traverses the IPsec boundary. With this **DataType**, the **SetData()** function is to set the SPD entry information, which may add one new entry, delete one existed entry or flush the whole database according to the parameter values. The corresponding **Data** is of type `EFI_IPSEC_SPD_DATA`.

**IPsecConfigDataTypeSad**

The IPsec Security Association Database (aka SAD) setting. A SA is a simplex connection that affords security services to the traffic carried by it. Security services are afforded to an SA by the use of AH, or ESP, but not both. The corresponding **Data** is of type `EFI_IPSEC_SA_DATA2` or `EFI_IPSEC_SAD_DATA`. Compared with `EFI_IPSEC_SA_DATA`, the `EFI_IPSEC_SA_DATA2` contains the extra Tunnel Source Address and Tunnel Destination Address thus it is recommended to be use if the implementation supports tunnel mode.

**IPsecConfigDataTypePad**

The IPsec Peer Authorization Database (aka PAD) setting, which provides the link between the SPD and a security association management protocol. The PAD entry specifies the authentication protocol (e.g. IKEv1, IKEv2) method used and the authentication data. The corresponding **Data** is of type `EFI_IPSEC_PAD_DATA`.
The `EFI_IPSEC_CONFIG_SELECTOR` describes the expected IPsec configuration data selector of type `EFI_IPSEC_CONFIG_DATA_TYPE`.

```c
typedef union {
    EFI_IPSEC_SPD_SELECTOR     SpdSelector;
    EFI_IPSEC_SA_ID            SaId;
    EFI_IPSEC_PAD_ID           PadId;
} EFI_IPSEC_CONFIG_SELECTOR;
```

LocalAddressCount specifies the actual number of entries in `LocalAddress`. A list of ranges of IPv4 or IPv6 addresses, which refers to the addresses being protected by IPsec policy.

RemoteAddressCount specifies the actual number of entries in `RemoteAddress`. A list of ranges of IPv4 or IPv6 addresses, which are peer entities to `LocalAddress`.

NextLayerProtocol Next layer protocol. Obtained from the IPv4 Protocol or the IPv6 Next Header fields. The next layer protocol is whatever comes after any IP extension headers that are present. A zero value is a wildcard that matches any value in `NextLayerProtocol` field.

LocalPort Local Port if the Next Layer Protocol uses two ports (as do TCP, UDP, and others). A zero value is a wildcard that matches any value in `LocalPort` field.
LocalPortRange  A designed port range size. The start port is LocalPort, and the total number of ports is described by LocalPortRange. This field is ignored if NextLayerProtocol does not use ports.

RemotePort  Remote Port if the Next Layer Protocol uses two ports. A zero value is a wildcard that matches any value in RemotePort field.

RemotePortRange  A designed port range size. The start port is RemotePort, and the total number of ports is described by RemotePortRange. This field is ignored if NextLayerProtocol does not use ports.

Note: The LocalPort and RemotePort selectors have different meaning depending on the NextLayerProtocol field. For example, if NextLayerProtocol value is ICMP, LocalPort and RemotePort describe the ICMP message type and code. This is described in section 4.4.1.1 of RFC 4301).

```c
//************************************************************
// EFI_IP_ADDRESS_INFO
//************************************************************
typedef struct _EFI_IP_ADDRESS_INFO {
    EFI_IP_ADDRESS Address;
    UINT8 PrefixLength;
} EFI_IP_ADDRESS_INFO;

#define MAX_PEERID_LEN   128
```

Name  A null-terminated ASCII name string which is used as a symbolic identifier for an IPsec Local or Remote address. The Name is optional, and can be NULL.

PackageFlag  Bit-mapped list describing Populate from Packet flags. When creating a SA, if PackageFlag bit is set to TRUE, instantiate the selector from the corresponding field in the package that
triggered the creation of the SA, else from the value(s) in the corresponding SPD entry. The PackageFlag bit setting for corresponding selector field of EFI_IPSEC_SPD_SELECTOR:
- Bit 0: EFI_IPSEC_SPD_SELECTOR.LocalAddress
- Bit 1: EFI_IPSEC_SPD_SELECTOR.RemoteAddress
- Bit 2: EFI_IPSEC_SPD_SELECTOR.NextLayerProtocol
- Bit 3: EFI_IPSEC_SPD_SELECTOR.LocalPort
- Bit 4: EFI_IPSEC_SPD_SELECTOR.RemotePort
Others: Reserved.

TrafficDirection
The traffic direction of datagram.

Action
Processing choices to indicate which action is required by this policy.

ProcessingPolicy
The policy and rule information for a SPD entry. The type EFI_IPSEC_PROCESSPOLICY is defined in below.

SaIdCount
Specifies the actual number of entries in SaId list.

SaId
Pointer to the SAD entry used for the traffic processing. The existed SAD entry links indicate this is the manual key case.

//************************************************************
// EFI_IPSEC_TRAFFIC_DIR
//************************************************************
typedef enum {
  EfiIPsecInBound,
  EfiIPsecOutBound
} EFI_IPSEC_TRAFFIC_DIR;

The EFI_IPSEC_TRAFFIC_DIR represents the directionality in an SPD entry. The EfiIPsecInBound refers to traffic entering an IPsec implementation via the unprotected interface or emitted by the implementation on the unprotected side of the boundary and directed towards the protected interface. The EfiIPsecOutBound refers to traffic entering the implementation via the protected interface, or emitted by the implementation on the protected side of the boundary and directed toward the unprotected interface.

//************************************************************
// EFI_IPSEC_ACTION
//************************************************************
typedef enum {
  EfiIPsecActionDiscard,
  EfiIPsecActionBypass,
  EfiIPsecActionProtect
} EFI_IPSEC_ACTION;

For any inbound or outbound datagram, EFI_IPSEC_ACTION represents three possible processing choices:
EfiIPsecActionDiscard
Refers to traffic that is not allowed to traverse the IPsec boundary (in the direction specified by EFi_IPSEC_TRAFFIC_DIR);

EfiIPsecActionByPass
Refers to traffic that is allowed to cross the IPsec boundary without protection.

EfiIPsecActionProtect
Refers to traffic that is afforded IPsec protection, and for such traffic the SPD must specify the security protocols to be employed, their mode, security service options, and the cryptographic algorithms to be used.

//EFI_IPSEC_PROCESS_POLICY
typedef struct _EFI_IPSEC_PROCESS_POLICY {
    BOOLEAN ExtSeqNum;
    BOOLEAN SeqOverflow;
    BOOLEAN FragCheck;
    EFI_IPSEC_SA_LIFETIME SaLifetime;
    EFI_IPSEC_MODE Mode;
    EFI_IPSEC_TUNNEL_OPTION *TunnelOption;
    EFI_IPSEC_PROTOCOL_TYPE Proto;
    UINT8 AuthAlgoId;
    UINT8 EncAlgoId;
} EFI_IPSEC_PROCESS_POLICY;

If required action of an SPD entry is EfiIPsecActionProtect, the EFI_IPSEC_PROCESS_POLICY structure describes a policy list for traffic processing.

- **ExtSeqNum**
  Extended Sequence Number. Is this SA using extended sequence numbers. 64 bit counter is used if TRUE.

- **SeqOverflow**
  A flag indicating whether overflow of the sequence number counter should generate an auditable event and prevent transmission of additional packets on the SA, or whether rollover is permitted.

- **FragCheck**
  Is this SA using stateful fragment checking. TRUE represents stateful fragment checking.

- **SaLifetime**
  A time interval after which a SA must be replaced with a new SA (and new SPI) or terminated. The type EFI_IPSEC_SA_LIFETIME is defined in below.

- **Mode**
  IPsec mode: tunnel or transport
TunnelOption

Tunnel Option. TunnelOption is ignored if Mode is EfiIPsecTransport. The type EFI_IPSEC_TUNNEL_OPTION is defined in below

Proto

IPsec protocol: AH or ESP

AuthAlgoId

Cryptographic algorithm type used for authentication

EncAlgoId

Cryptographic algorithm type used for encryption. EncAlgo is NULL when IPsec protocol is AH. For ESP protocol, EncAlgo can also be used to describe the algorithm if a combined mode algorithm is used.

// EFII_IPSEC_SA_LIFETIME
typedef struct _EFI_IPSEC_SA_LIFETIME {
  UINT64 ByteCount;
  UINT64 SoftLifetime;
  UINT64 HardLifetime
} EFI_IPSEC_SA_LIFETIME;

 EFI_IPSEC_SA_LIFETIME defines the lifetime of an SA, which represents when a SA must be replaced or terminated. A value of all 0 for each field removes the limitation of a SA lifetime.

  ByteCount
  The number of bytes to which the IPsec cryptographic algorithm can be applied. For ESP, this is the encryption algorithm and for AH, this is the authentication algorithm. The ByteCount includes pad bytes for cryptographic operations.

  SoftLifetime
  A time interval in second that warns the implementation to initiate action such as setting up a replacement SA.

  HardLifetime
  A time interval in second when the current SA ends and is destroyed.

// EFII_IPSEC_MODE
typedef enum {
  EfiIPsecTransport,
  EfiIPsecTunnel
} EFI_IPSEC_MODE;

There are two modes of IPsec operation: transport mode and tunnel mode. In EfiIPsecTransport mode, AH and ESP provide protection primarily for next layer protocols; In EfiIPsecTunnel mode, AH and ESP are applied to tunneled IP packets.
typedef enum {
    EfiIPsecTunnelClearDf,
    EfiIPsecTunnelSetDf,
    EfiIPsecTunnelCopyDf
} EFI_IPSEC_TUNNEL_DF_OPTION;

The option of copying the DF bit from an outbound package to the tunnel mode header that it emits, when traffic is carried via a tunnel mode SA. This applies to SAs where both inner and outer headers are IPv4. The value can be:

**EfiIPsecTunnelClearDf**: Clear DF bit from inner header

**EfiIPsecTunnelSetDf**: Set DF bit from inner header

**EfiIPsecTunnelCopyDf**: Copy DF bit from inner header

typedef struct _ EFI_IPSEC_TUNNEL_OPTION {
    EFI_IP_ADDRESS LocalTunnelAddress;
    EFI_IP_ADDRESS RemoteTunnelAddress;
    EFI_IPSEC_TUNNEL_DF_OPTION DF;
} EFI_IPSEC_TUNNEL_OPTION;

LocalTunnelAddress Local tunnel address when IPsec mode is EfiIPsecTunnel
RemoteTunnelAddress Remote tunnel address when IPsec mode is EfiIPsecTunnel
DF The option of copying the DF bit from an outbound package to the tunnel mode header that it emits, when traffic is carried via a tunnel mode SA.

typedef enum {
    EfiIPsecAH,
    EfiIPsecESP
} EFI_IPSEC_PROTOCOL_TYPE;

IPsec protocols definition. **EfiIPsecAH** is the IP Authentication Header protocol which is specified in RFC 4302. **EfiIPsecESP** is the IP Encapsulating Security Payload which is specified in RFC 4303.
typedef struct _EFI_IPSEC_SA_ID {
  UINT32 Spi;
  EFI_IPSEC_PROTOCOL_TYPE Proto;
  EFI_IP_ADDRESS DestAddress;
} EFI_IPSEC_SA_ID;

A triplet to identify an SA, consisting of the following members:

Spi Security Parameter Index (aka SPI). An arbitrary 32-bit value that is used by a receiver to identify the SA to which an incoming package should be bound.

Proto IPsec protocol: AH or ESP

DestAddress Destination IP address.

typedef struct _EFI_IPSEC_SA_DATA {
  EFI_IPSEC_MODE Mode;
  UINT64 SNCount;
  UINT8 AntiReplayWindows;
  EFI_IPSEC_ALGO_INFO AlgoInfo;
  EFI_IPSEC_SA_LIFETIME SaLifetime;
  UINT32 PathMTU;
  EFI_IPSEC_SPD_SELECTOR *SpdSelector;
  BOOLEAN ManualSet
} EFI_IPSEC_SA_DATA;

The data items defined in one SAD entry:

Mode IPsec mode: tunnel or transport

SNCount Sequence Number Counter. A 64-bit counter used to generate the sequence number field in AH or ESP headers.

ReplayWindows Anti-Replay Window. A 64-bit counter and a bit-map used to determine whether an inbound AH or ESP packet is a replay.

AlgoInfo AH/ESP cryptographic algorithm, key and parameters.

SaLifeTime Lifetime of this SA.

PathMTU Any observed path MTU and aging variables. The Path MTU processing is defined in section 8 of RFC 4301.

SpdSelector Link to one SPD entry.
ManualSet: Indication of whether it's manually set or negotiated automatically. If ManualSet is FALSE, the corresponding SA entry is inserted through IKE protocol negotiation.

---

```c
/*EFI_IPSEC_SA_DATA2*/
typedef struct _EFI_IPSEC_SA_DATA2 {
    EFI_IPSEC_MODE Mode;
    UINT64 SNCount;
    UINT8 AntiReplayWindows;
    EFI_IPSEC_ALGO_INFO AlgoInfo;
    EFI_IPSEC_SA_LIFETIME SaLifetime;
    UINT32 PathMTU;
    EFI_IPSEC_SPD_SELECTOR *SpdSelector;
    BOOLEAN ManualSet;
    EFI_IP_ADDRESS TunnelSourceAddress;
    EFI_IP_ADDRESS TunnelDestinationAddress
} EFI_IPSEC_SA_DATA2;
```

The data items defined in one SAD entry:

- **Mode**: IPsec mode: tunnel or transport.
- **SNCount**: Sequence Number Counter. A 64-bit counter used to generate the sequence number field in AH or ESP headers.
- **ReplayWindows**: Anti-Replay Window. A 64-bit counter and a bit-map used to determine whether an inbound AH or ESP packet is a replay.
- **AlgoInfo**: AH/ESP cryptographic algorithm, key and parameters.
- **SaLifeTime**: Lifetime of this SA.
- **PathMTU**: Any observed path MTU and aging variables. The Path MTU processing is defined in section 8 of RFC 4301.
- **SpdSelector**: Link to one SPD entry.
- **ManualSet**: Indication of whether it's manually set or negotiated automatically. If ManualSet is FALSE, the corresponding SA entry is inserted through IKE protocol negotiation.
- **TunnelSourceAddress**: The tunnel header IP source address.
- **TunnelDestinationAddress**: The tunnel header IP destination address.
The security algorithm selection for IPsec AH authentication. The required authentication algorithm is specified in RFC 4305.

The security algorithm selection for IPsec ESP encryption and authentication. The required authentication algorithm is specified in RFC 4305. EncAlgoId fields can also specify an ESP combined mode algorithm (e.g. AES with CCM mode, specified in RFC 4309), which provides both confidentiality and authentication services.
typedef struct _EFI_IPSEC_PAD_ID {
    BOOLEAN          PeerIdValid;
    union {
        EFI_IP_ADDRESS_INFO   IpAddress;
        UINT8                PeerId [MAX_PEERID_LEN];
    } Id;
} EFI_IPSEC_PAD_ID;

The entry selector for IPsec PAD that represents how to authenticate each peer. EFI_IPSEC_PAD_ID specifies the identifier for PAD entry, which is also used for SPD lookup.

IpAddress    Pointer to the IPv4 or IPv6 address range.
PeerId        Pointer to a null-terminated ASCII string representing the symbolic names. A PeerId can be a DNS name, Distinguished Name, RFC 822 email address or Key ID (specified in section 4.4.3.1 of RFC 4301)

typedef struct _EFI_IPSEC_PAD_DATA {
    EFI_IPSEC_AUTH_PROTOCOL_TYPE AuthProtocol;
    EFI_IPSEC_AUTH_METHOD        AuthMethod;
    BOOLEAN                     IkeIdFlag;
    UINTN                       AuthDataSize;
    VOID                        *AuthData;
    UINTN                       RevocationDataSize;
    VOID                        *RevocationData;
} EFI_IPSEC_PAD_DATA;

The data items defined in one PAD entry:

AuthProtocol    Authentication Protocol for IPsec security association management
AuthMethod      Authentication method used.
IkeIdFlag       The IKE ID payload will be used as a symbolic name for SPD lookup if IkeIdFlag is TRUE. Otherwise, the remote IP address provided in traffic selector payloads will be used.
AuthDataSize    The size of Authentication data buffer, in bytes.
AuthData        Buffer for Authentication data, (e.g., the pre-shared secret or the trust anchor relative to which the peer's certificate will be validated).
RevocationDataSize The size of RevocationData, in bytes.
RevocationData Pointer to CRL or OCSP data, if certificates are used for authentication method.

//***************************************************************
// EFI_IPSEC_AUTH_PROTOCOL
//***************************************************************
typedef enum {
    EfiIPsecAuthProtocolIKEv1,
    EfiIPsecAuthProtocolIKEv2,
    EfiIPsecAuthProtocolMaximum
} EFI_IPSEC_AUTH_PROTOCOL_TYPE;

EFI_IPSEC_AUTH_PROTOCOL_TYPE defines the possible authentication protocol for IPsec security association management.

//***************************************************************
// EFI_IPSEC_AUTH_METHOD
//***************************************************************
typedef enum {
    EfiIPsecAuthMethodPreSharedSecret,
    EfiIPsecAuthMethodCertificates,
    EfiIPsecAuthMethodMaximum
} EFI_IPSEC_AUTH_METHOD;

EfiIPsecAuthMethodCertificates IKE employs X.509 certificates for SA establishment.

Status Codes Returned

<table>
<thead>
<tr>
<th>Code</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>EFI_SUCCESS</td>
<td>The specified configuration entry data is set successfully.</td>
</tr>
</tbody>
</table>
| EFI_INVALID_PARAMETER         | One or more of the following are TRUE:
|                               | • This is NULL.                                                            |
| EFI_UNSUPPORTED               | The specified DataType is not supported.                                   |
| EFI_OUT_OF_RESOURCES          | The required system resource could not be allocated.                      |

EFI_IPSEC_CONFIG_PROTOCOL.GetData()

Summary
Return the configuration value for the EFI IPsec driver.
Prototype

typedef
   EFI_STATUS
   (EFIAPI *EFI_IPSEC_CONFIG_GET_DATA) (
      IN EFI_IPSEC_CONFIG_PROTOCOL    *This,
      IN EFI_IPSEC_CONFIG_DATA_TYPE   DataType,
      IN EFI_IPSEC_CONFIG_SELECTOR    *Selector,
      IN OUT UINTN            *DataSize,
      OUT VOID              *Data
    );

Parameters

This    Pointer to the EFI_IPSEC_CONFIG_PROTOCOL instance.
DataType The type of data to retrieve. Type EFI_IPSEC_CONFIG_DATA_TYPE is defined in EFI_IPSEC_CONFIG_PROTOCOL.SetData().
Selector Pointer to an entry selector which is an identifier of the IPsec configuration data entry. Type EFI_IPSEC_CONFIG_SELECTOR is defined in the EFI_IPSEC_CONFIG_PROTOCOL.SetData() function description.
DataSize On output the size of data returned in Data.
Data     The buffer to return the contents of the IPsec configuration data. The type of the data buffer is associated with the DataType.

Description

This function lookup the data entry from IPsec database or IKEv2 configuration information. The expected data type and unique identification are described in DataType and Selector parameters.

Status Codes Returned

<table>
<thead>
<tr>
<th>Status Code</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>EFI_SUCCESS</td>
<td>The specified configuration data is got successfully.</td>
</tr>
<tr>
<td>EFI_INVALID_PARAMETER</td>
<td>One or more of the followings are TRUE:</td>
</tr>
<tr>
<td></td>
<td>• This is NULL</td>
</tr>
<tr>
<td></td>
<td>• Selector is NULL</td>
</tr>
<tr>
<td></td>
<td>• DataSize is NULL</td>
</tr>
<tr>
<td></td>
<td>• Data is NULL</td>
</tr>
<tr>
<td>EFI_NOT_FOUND</td>
<td>The configuration data specified by Selector is not found.</td>
</tr>
<tr>
<td>EFI_UNSUPPORTED</td>
<td>The specified DataType is not supported.</td>
</tr>
<tr>
<td>EFI_BUFFER_TOO_SMALL</td>
<td>The DataSize is too small for the result. DataSize has been updated with the size needed to complete the request.</td>
</tr>
</tbody>
</table>
EFI_IPSEC_CONFIG_PROTOCOL.GetNextSelector()

Summary
Enumerates the current selector for IPsec configuration data entry.

Prototype
```c
typedef EFI_STATUS
    (EFIAPI *EFI_IPSEC_CONFIG_GET_NEXT_SELECTOR) (
    IN EFI_IPSEC_CONFIG_PROTOCOL    *This,
    IN EFI_IPSEC_CONFIG_DATA_TYPE    DataType,
    IN OUT UINTN            *SelectorSize,
    IN OUT EFI_IPSEC_CONFIG_SELECTOR  *Selector,
    );
```

Parameters
- **This**: Pointer to the EFI_IPSEC_CONFIG_PROTOCOL instance.
- **DataType**: The type of IPsec configuration data to retrieve. Type EFI_IPSEC_CONFIG_DATA_TYPE is defined in EFI_IPSEC_CONFIG_PROTOCOL.SetData().
- **SelectorSize**: The size of the Selector buffer.
- **Selector**: On input, supplies the pointer to last Selector that was returned by GetNextSelector(). On output, returns one copy of the current entry Selector of a given DataType. Type EFI_IPSEC_CONFIG_SELECTOR is defined in the EFI_IPSEC_CONFIG_PROTOCOL.SetData() function description.

Description
This function is called multiple times to retrieve the entry Selector in IPsec configuration database. On each call to GetNextSelector(), the next entry Selector are retrieved into the output interface. If the entire IPsec configuration database has been iterated, the error EFI_NOT_FOUND is returned. If the Selector buffer is too small for the next Selector copy, an EFI_BUFFER_TOO_SMALL error is returned, and SelectorSize is updated to reflect the size of buffer needed.

On the initial call to GetNextSelector() to start the IPsec configuration database search, a pointer to the buffer with all zero value is passed in Selector. Calls to SetData() between calls to GetNextSelector may produce unpredictable results.
**Status Codes Returned**

<table>
<thead>
<tr>
<th>Code</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>EFI_SUCCESS</td>
<td>The specified configuration data is got successfully.</td>
</tr>
<tr>
<td>EFI_INVALID_PARAMETER</td>
<td>One or more of the followings are <strong>TRUE</strong>:</td>
</tr>
<tr>
<td></td>
<td>• This is <strong>NULL</strong></td>
</tr>
<tr>
<td></td>
<td>• SelectorSize is <strong>NULL</strong></td>
</tr>
<tr>
<td></td>
<td>• Selector is <strong>NULL</strong></td>
</tr>
<tr>
<td>EFI_NOT_FOUND</td>
<td>The next configuration data entry was not found.</td>
</tr>
<tr>
<td>EFI_UNSUPPORTED</td>
<td>The specified <strong>DataType</strong> is not supported.</td>
</tr>
<tr>
<td>EFI_BUFFER_TOO_SMALL</td>
<td>The <strong>SelectorSize</strong> is too small for the result. This parameter has been updated with the size needed to complete the search request.</td>
</tr>
</tbody>
</table>

**EFI_IPSEC_CONFIG_PROTOCOL.RegisterDataNotify ()**

**Summary**

Register an event that is to be signaled whenever a configuration process on the specified IPsec configuration information is done.

**Prototype**

```c
typedef EFI_STATUS (EFIAPI *EFI_IPSEC_CONFIG_REGISTER_NOTIFY) (  
    IN EFI_IPSEC_CONFIG_PROTOCOL    *This,  
    IN EFI_IPSEC_CONFIG_DATA_TYPE    DataType,  
    IN EFI_EVENT                      Event );
```

**Parameters**

- **This**
  - Pointer to the **EFI_IPSEC_CONFIG_PROTOCOL** instance.
- **DataType**
  - The type of data to be registered the event for. Type **EFI_IPSEC_CONFIG_DATA_TYPE** is defined in **EFI_IPSEC_CONFIG_PROTOCOL.SetData()** function description.
- **Event**
  - The event to be registered.

**Description**

This function registers an event that is to be signaled whenever a configuration process on the specified IPsec configuration data is done (e.g. IPsec security policy database configuration is ready). An event can be registered for different **DataType** simultaneously and the caller is responsible for determining which type of configuration data causes the signaling of the event in such case.
Status Codes Returned

<table>
<thead>
<tr>
<th>Code</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>EFI_SUCCESS</td>
<td>The event is registered successfully.</td>
</tr>
<tr>
<td>EFI_INVALID_PARAMETER</td>
<td>This is NULL or Event is NULL.</td>
</tr>
<tr>
<td>EFI_ACCESS_DENIED</td>
<td>The Event is already registered for the DataType.</td>
</tr>
<tr>
<td>EFI_UNSUPPORTED</td>
<td>The notify registration unsupported or the specified DataType is not</td>
</tr>
<tr>
<td></td>
<td>supported.</td>
</tr>
</tbody>
</table>

EFI_IPSEC_CONFIG_PROTOCOL.UnregisterDataNotify()

Summary
Remove the specified event that is previously registered on the specified IPsec configuration data.

Prototype

typedef
    EFI_STATUS
    (EFIAPI *EFI_IPSEC_CONFIG_UNREGISTER_NOTIFY) (  
        IN EFI_IPSEC_CONFIG_PROTOCOL    *This,  
        IN EFI_IPSEC_CONFIG_DATA_TYPE   DataType,  
        IN EFI_EVENT                   Event
    );

Parameters

This Pointer to the EFI_IPSEC_CONFIG_PROTOCOL instance.

DataType The configuration data type to remove the registered event for. Type EFI_IPSEC_CONFIG_DATA_TYPE is defined in EFI_IPSEC_CONFIG_PROTOCOL.SetData() function description.

Event The event to be unregistered.

Description
This function removes a previously registered event for the specified configuration data.

Status Codes Returned

<table>
<thead>
<tr>
<th>Code</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>EFI_SUCCESS</td>
<td>The event is removed successfully.</td>
</tr>
<tr>
<td>EFI_NOT_FOUND</td>
<td>The Event specified by DataType could not be found in the database.</td>
</tr>
<tr>
<td>EFI_INVALID_PARAMETER</td>
<td>This is NULL or Event is NULL.</td>
</tr>
<tr>
<td>EFI_UNSUPPORTED</td>
<td>The notify registration unsupported or the specified DataType is not</td>
</tr>
<tr>
<td></td>
<td>supported.</td>
</tr>
</tbody>
</table>

28.8.3 EFI IPsec Protocol
This section provides a detailed description of the EFI_IPSEC_PROTOCOL. This protocol handles IPsec-protected traffic.
**EFI_IPSEC_PROTOCOL**

**Summary**
The **EFI_IPSEC_PROTOCOL** is used to abstract the ability to deal with the individual packets sent and received by the host and provide packet-level security for IP datagram.

**GUID**
```c
#define EFI_IPSEC_PROTOCOL_GUID {
{0xdfb386f7,0xe100,0x43ad,
{0x9c,0x9a,0xed,0x90,0xd0,0x8a,0x5e,0x12 }}
```

**Protocol Interface Structure**
```c
typedef struct _EFI_IPSEC_PROTOCOL {
  EFI_IPSEC_PROCESS  Process;
  EFI_EVENT DisabledEvent;
  BOOLEAN DisabledFlag;
} EFI_IPSEC_PROTOCOL;
```

**Parameters**
- **Process** Handle the IPsec message.
- **DisabledEvent** Event signaled when the interface is disabled.
- **DisabledFlag** State of the interface.

**Description**
The **EFI_IPSEC_PROTOCOL** provides the ability for securing IP communications by authenticating and/or encrypting each IP packet in a data stream.

**EFI_IPSEC_PROTOCOL** can be consumed by both the IPv4 and IPv6 stack. A user can employ this protocol for IPsec package handling in both IPv4 and IPv6 environment.

**EFI_IPSEC_PROTOCOL.Process()**

**Summary**
Handles IPsec packet processing for inbound and outbound IP packets.
Prototype

typedef
EFI_STATUS
(EFI_API *EFI_IPSEC_PROCESS) (
    IN EFI_IPSEC_PROTOCOL     *This,
    IN EFI_HANDLE              NicHandle,
    IN UINT8                    IpVer,
    IN OUT VOID                 *IpHead,
    IN UINT8                    *LastHead,
    IN VOID                     *OptionsBuffer,
    IN UINT32                   OptionsLength,
    IN OUT EFI_IPSEC_FRAGMENT_DATA **FragmentTable,
    IN UINT32                   *FragmentCount,
    IN EFI_IPSEC_TRAFFIC_DIR     TrafficDirection,
    OUT EFI_EVENT               *RecycleSignal
)

Related definitions

//***********************************************************
// EFI_IPSEC_FRAGMENT_DATA //
***********************************************************
typedef struct _EFI_IPSEC_FRAGMENT_DATA {
    UINT32 FragmentLength;
    VOID  *FragmentBuffer;
} EFI_IPSEC_FRAGMENT_DATA;

EFI_IPSEC_FRAGMENT_DATA defines the instances of packet fragments.

Parameters

This            Pointer to the EFI_IPSEC_PROTOCOL instance.
NicHandle       Instance of the network interface.
IpVer           IPV4 or IPV6.
IpHead          Pointer to the IP Header.
LastHead        The protocol of the next layer to be processed by IPsec.
OptionsBuffer   Pointer to the options buffer.
OptionsLength   Length of the options buffer.
FragmentTable   Pointer to a list of fragments.
FragmentCount   Number of fragments.
TrafficDirection Traffic direction.
RecycleSignal   Event for recycling of resources.
Description

The **EFI_IPSEC_PROCESS** process routine handles each inbound or outbound packet. The behavior is that it can perform one of the following actions: bypass the packet, discard the packet, or protect the packet.

**Status Codes Returned**

<table>
<thead>
<tr>
<th>Code</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>EFI_SUCCESS</td>
<td>The packet was bypassed and all buffers remain the same.</td>
</tr>
<tr>
<td>EFI_SUCCESS</td>
<td>The packet was protected.</td>
</tr>
<tr>
<td>EFI_ACCESS_DENIED</td>
<td>The packet was discarded.</td>
</tr>
</tbody>
</table>

28.8.4 EFI IPsec2 Protocol

This section provides a detailed description of the **EFI_IPSEC2_PROTOCOL**. This protocol handles IPsec-protected traffic.

**EFI_IPSEC2_PROTOCOL**

**Summary**

The **EFI_IPSEC2_PROTOCOL** is used to abstract the ability to deal with the individual packets sent and received by the host and provide packet-level security for IP datagram.

**GUID**

```
#define EFI_IPSEC2_PROTOCOL_GUID \
{0xa3979e64, 0xace8, 0x4ddc, \
 {0xbc, 0x07, 0x4d, 0x66, 0xb8, 0xfd, 0x09, 0x77}};
```

**Protocol Interface Structure**

```
typedef struct _EFI_IPSEC2_PROTOCOL {
    EFI_IPSEC_PROCESSEXT ProcessExt;
    EFI_EVENT DisabledEvent;
    BOOLEAN DisabledFlag;
} EFI_IPSEC2_PROTOCOL;
```

**Parameters**

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>ProcessExt</td>
<td>Handle the IPsec message with the extension header processing support.</td>
</tr>
<tr>
<td>DisabledEvent</td>
<td>Event signaled when the interface is disabled.</td>
</tr>
<tr>
<td>DisabledFlag</td>
<td>State of the interface.</td>
</tr>
</tbody>
</table>

**Description**

The **EFI_IPSEC2_PROTOCOL** provides the ability for securing IP communications by authenticating and/or encrypting each IP packet in a data stream.

**EFI_IPSEC2_PROTOCOL** can be consumed by both the IPv4 and IPv6 stack. A user can employ this protocol for IPsec package handling in both IPv4 and IPv6 environment.
EFI_IPSEC2_PROTOCOL.ProcessExt()

Summary
Handles IPsec processing for both inbound and outbound IP packets. Compare with Process() in EFI_IPSEC_PROTOCOL, this interface has the capability to process Option(Extension Header).

Prototype
Typedef
EFI_STATUS
(EFI_API *EFI_IPSEC_PROCESSEX T) (  
IN EFI_IPSEC2_PROTOCOL *This,  
IN EFI_HANDLE NicHandle,  
IN UINT8 IpVer,  
IN OUT VOID *IpHead,  
IN OUT UINT8 *LastHead,  
IN OUT VOID **OptionsBuffer,  
IN OUT UINT8 *OptionsLength,  
IN OUT EFI_IPSEC_FRAGMENT_DATA **FragmentTable,  
IN OUT UINT32 *FragmentCount,  
IN EFI_IPSEC_TRAFFIC_DIR TrafficDirection,  
OUT EFI_EVENT *RecycleSignal  
)

Parameters
This
Pointer to the EFI_IPSEC2_PROTOCOL instance.
NicHandle
Instance of the network interface.
IpVer
IP version.IPV4 or IPV6.
IpHead
Pointer to the IP Header it is either the EFI_IP4_HEADER or EFI_IP6_HEADER. On input, it contains the IP header. On output,
1) in tunnel mode and the traffic direction is inbound, the buffer will be reset to zero by IPsec;
2) in tunnel mode and the traffic direction is outbound, the buffer will reset to be the tunnel IP header.
3) in transport mode, the related fielders (like payload length, Next header) in IP header will be modified according to the condition.
LastHead
For IP4, it is the next protocol in IP header. For IP6 it is the Next Header of the last extension header.
OptionsBuffer
On input, it contains the options (extensions header) to be processed by IPsec. On output,
1) in tunnel mode and the traffic direction is outbound, it will be set to NULL, and that means this contents was wrapped
after inner header and should not be concatenated after tunnel header again;

2) in transport mode and the traffic direction is inbound, if there are IP options (extension headers) protected by IPsec, IPsec will concatenate the those options after the input options (extension headers);

3) on other situations, the output of contents of 
\texttt{OptionsBuffer} might be same with input's. The caller should take the responsibility to free the buffer both on input and on output.

\begin{itemize}
  \item \textbf{OptionsLength} On input, the input length of the options buffer. On output, the output length of the options buffer.
  \item \textbf{FragmentTable} Pointer to a list of fragments. On input, these fragments contain the IP payload. On output,
    \begin{itemize}
      \item 1) in tunnel mode and the traffic direction is inbound, the fragments contain the whole IP payload which is from the IP inner header to the last byte of the packet;
      \item 2) in tunnel mode and the traffic direction is the outbound, the fragments contains the whole encapsulated payload which encapsulates the whole IP payload between the encapsulated header and encapsulated trailer fields.
      \item 3) in transport mode and the traffic direction is inbound, the fragments contains the IP payload which is from the next layer protocol to the last byte of the packet;
      \item 4) in transport mode and the traffic direction is outbound, the fragments contains the whole encapsulated payload which encapsulates the next layer protocol information between the encapsulated header and encapsulated trailer fields.
    \end{itemize}
  \item \textbf{FragmentCount} Number of fragments.
  \item \textbf{TrafficDirection} Traffic direction.
  \item \textbf{RecycleSignal} Event for recycling of resources.
\end{itemize}

\textbf{Description}

The \texttt{EFI_IPSEC_PROCESSEXT} process routine handles each inbound or outbound packet with the support of options (extension headers) processing. The behavior is that it can perform one of the following actions: bypass the packet, discard the packet, or protect the packet.
28.9 Network Protocol - EFI FTP Protocol

This section defines the EFI FTPv4 (File Transfer Protocol version 4) Protocol that interfaces over EFI FTPv4 Protocol

**EFI_FTP4_SERVICE_BINDING_PROTOCOL Summary**

**Summary**

The **EFI_FTP4_SERVICE_BINDING_PROTOCOL** is used to locate communication devices that are supported by an EFI FTPv4 Protocol driver and to create and destroy instances of the EFI FTPv4 Protocol child protocol driver that can use the underlying communication device.

**GUID**

```c
#define EFI_FTP4_SERVICE_BINDING_PROTOCOL_GUID \
    {0xfaaecb1, 0x226e, 0x4782, \ 
     {0xaa, 0xce, 0x7d, 0xb9, 0xbc, 0xbf, 0x4d, 0xaf}}
```

**Description**

A network application or driver that requires FTPv4 I/O services can use one of the protocol handler services, such as **BS->LocateHandleBuffer()**, to search for devices that publish an EFI FTPv4 Service Binding Protocol GUID. Each device with a published EFI FTPv4 Service Binding Protocol GUID supports the EFI FTPv4 Protocol service and may be available for use.

After a successful call to the **EFI_FTP4_SERVICE_BINDING_PROTOCOL.CreateChild()** function, the newly created child EFI FTPv4 Protocol driver instance is in an unconfigured state; it is not ready to transfer data.

Before a network application terminates execution, every successful call to the **EFI_FTP4_SERVICE_BINDING_PROTOCOL.CreateChild()** function must be matched with a call to the **EFI_FTP4_SERVICE_BINDING_PROTOCOL.DestroyChild()** function.

Each instance of the EFI FTPv4 Protocol driver can support one file transfer operation at a time. To download two files at the same time, two instances of the EFI FTPv4 Protocol driver will need to be created.

<table>
<thead>
<tr>
<th>Status Code Returned</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>EFI_SUCCESS</td>
<td>The packet was bypassed and all buffers remain the same.</td>
</tr>
<tr>
<td>EFI_SUCCESS</td>
<td>The packet was processed by IPsec successfully.</td>
</tr>
<tr>
<td>EFI_ACCESS_DENIED</td>
<td>The packet was discarded.</td>
</tr>
<tr>
<td>EFI_NOT_READY</td>
<td>The IKE negotiation is invoked and the packet was discarded.</td>
</tr>
<tr>
<td>EFI_INVALID_PARAMETER</td>
<td>One of more of following are TRUE:</td>
</tr>
<tr>
<td></td>
<td>If OptionsBuffer is <strong>NULL</strong>;</td>
</tr>
<tr>
<td></td>
<td>If OptionsLength is <strong>NULL</strong>;</td>
</tr>
<tr>
<td></td>
<td>If FragmentTable is <strong>NULL</strong>;</td>
</tr>
<tr>
<td></td>
<td>If FragmentCount is <strong>NULL</strong>;</td>
</tr>
</tbody>
</table>
Note: Byte Order: if not specifically specified, the IP addresses used in the EFI_FTP4_PROTOCOL are in network byte order and the ports are in host byte order.

EFI_FTP4_PROTOCOL

Summary
The EFI FTPv4 Protocol provides basic services for client-side FTP (File Transfer Protocol) operations.

GUID
#define EFI_FTP4_PROTOCOL_GUID \
{0xeb338826, 0x681b, 0x4295, \
{0xb3, 0x56, 0x2b, 0x36, 0x4c, 0x75, 0x7b, 0x09}}

Protocol Interface Structure
typedef struct _EFI_FTP4_PROTOCOL {
  EFI_FTP4_GET_MODE_DATA  GetModeData;
  EFI_FTP4_CONNECT        Connect;
  EFI_FTP4_CLOSE          Close;
  EFI_FTP4_CONFIGURE      Configure;
  EFI_FTP4_READ_FILE      ReadFile;
  EFI_FTP4_WRITE_FILE     WriteFile;
  EFI_FTP4_READ_DIRECTORY ReadDirectory;
  EFI_FTP4_POLL           Poll;
} EFI_FTP4_PROTOCOL;

Parameters
GetModeData Reads the current operational settings. See the GetModeData() function description.
Connect Establish control connection with the FTP server by using the TELNET protocol according to FTP protocol definition. See the Connect() function description.
Close Gracefully disconnecting a FTP control connection. This function is a nonblocking operation. See the Close() function description.
Configure Sets and clears operational parameters for an FTP child driver. See the Configure() function description.
ReadFile Downloads a file from an FTPv4 server. See the ReadFile() function description.
WriteFile Uploads a file to an FTPv4 server. This function may be unsupported in some EFI implementations. See the WriteFile() function description.
ReadDirectory Download a related file "directory" from an FTPv4 server. This function may be unsupported in some implementations. See the ReadDirectory() function description.
Poll Polls for incoming data packets and processes outgoing data packets. See the Poll() function description.
**EFI_FTP4_PROTOCOL.GetModeData()**

**Summary**
Gets the current operational settings

**Prototype**
```c
typedef
  EFI_STATUS
  (EFIAPI *EFI_FTP4_GET_MODE_DATA)(
    IN EFI_FTP4_PROTOCOL    *This,
    OUT EFI_FTP4_CONFIG_DATA  *ModeData
  );
```

**Parameters**
- **This**
  Pointer to the EFI_FTP4_PROTOCOL instance.
- **ModeData**
  Pointer to storage for the EFI FTPv4 Protocol driver mode data. Type EFI_FTP4_CONFIG_DATA is defined in "Related Definitions" below. The string buffers for Username and Password in EFI_FTP4_CONFIG_DATA are allocated by the function, and the caller should take the responsibility to free the buffer later.

**Description**
The GetModeData() function reads the current operational settings of this EFI FTPv4 Protocol driver instance. EFI_FTP4_CONFIG_DATA is defined in the EFI_FTP4_PROTOCOL.Configure.

**Status Codes Returned**

<table>
<thead>
<tr>
<th>Status Code</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>EFI_SUCCESS</td>
<td>This function is called successfully.</td>
</tr>
<tr>
<td>EFI_INVALID_PARAMETER</td>
<td>One or more of the following are TRUE:</td>
</tr>
<tr>
<td></td>
<td>• This is NULL.</td>
</tr>
<tr>
<td></td>
<td>• ModeData is NULL.</td>
</tr>
<tr>
<td>EFI_NOT_STARTED</td>
<td>The EFI FTPv4 Protocol driver has not been started.</td>
</tr>
<tr>
<td>EFI_OUT_OF_RESOURCES</td>
<td>Could not allocate enough resource to finish the operation.</td>
</tr>
<tr>
<td>EFI_DEVICE_ERROR</td>
<td>An unexpected system or network error occurred.</td>
</tr>
</tbody>
</table>

**EFI_FTP4_PROTOCOL.Connect()**

**Summary**
Initiate a FTP connection request to establish a control connection with FTP server
Prototype

typedef
EFI_STATUS
(EIFIAPI *EFI_FTP4_CONNECT) (
    IN EFI_FTP4_PROTOCOL *This,
    IN EFI_FTP4_CONNECTION_TOKEN *Token
);

Parameters

This Pointer to the EFI_FTP4_PROTOCOL instance.
Token Pointer to the token used to establish control connection.

Related Definitions

//***************************************************************
// EFI_FTP4_CONNECTION_TOKEN
//***************************************************************
typedef struct {
    EFI_EVENT Event;
    EFI_STATUS Status;
} EFI_FTP4_CONNECTION_TOKEN;

Event

The Event to signal after the connection is established and Status field is updated by the EFI FTP v4 Protocol driver. The type of Event must be EVENT_NOTIFY_SIGNAL, and its Task Priority Level (TPL) must be lower than or equal to TPL_CALLBACK. If it is set to NULL, this function will not return until the function completes.

Status

The variable to receive the result of the completed operation.

Status Codes Returned

EFI_SUCCESS The FTP connection is established successfully.
EFI_ACCESS_DENIED The FTP server denied the access the user’s request to access it.
EFI_CONNECTION_RESET The connect fails because the connection is reset either by instance itself or communication peer.
EFI_TIMEOUT The connection establishment timer expired and no more specific information is available.
EFI_NETWORK_UNREACHABLE The active open fails because an ICMP network unreachable error is received.
EFI_HOST_UNREACHABLE The active open fails because an ICMP host unreachable error is received.
EFI_PROTOCOL_UNREACHABLE The active open fails because an ICMP protocol unreachable error is received.
EFI_PORT_UNREACHABLE The connection establishment timer times out and an ICMP port unreachable error is received.
The Connect() function will initiate a connection request to the remote FTP server with the corresponding connection token. If this function returns EFI_SUCCESS, the connection sequence is initiated successfully. If the connection succeeds or failed due to any error, the Token->Event will be signaled and Token->Status will be updated accordingly.

### Status Codes Returned

<table>
<thead>
<tr>
<th>Status Code</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>EFI_SUCCESS</td>
<td>The connection sequence is successfully initiated.</td>
</tr>
<tr>
<td>EFI_INVALID_PARAMETER</td>
<td>One or more of the following are TRUE:</td>
</tr>
<tr>
<td></td>
<td>² This is NULL.</td>
</tr>
<tr>
<td></td>
<td>² Token is NULL.</td>
</tr>
<tr>
<td></td>
<td>² Token-&gt;Event is NULL.</td>
</tr>
<tr>
<td>EFI_NOT_STARTED</td>
<td>The EFI FTPv4 Protocol driver has not been started.</td>
</tr>
<tr>
<td>EFI_NO_MAPPING</td>
<td>When using a default address, configuration (DHCP, BOOTP, RARP, etc.) is not finished yet.</td>
</tr>
<tr>
<td>EFI_OUT_OF_RESOURCES</td>
<td>Could not allocate enough resource to finish the operation.</td>
</tr>
<tr>
<td>EFI_DEVICE_ERROR</td>
<td>An unexpected system or network error occurred.</td>
</tr>
</tbody>
</table>

### EFI_FTP4_PROTOCOL.Close()

#### Summary
Disconnecting a FTP connection gracefully.

#### Prototype
```c
typedef EFI_STATUS (EFIAPI *EFI_FTP4_CLOSE)(
    IN EFI_FTP4_PROTOCOL       *This,
    IN EFI_FTP4_CONNECTION_TOKEN   *Token
);
```

#### Parameters
- **This** Pointer to the EFI_FTP4_PROTOCOL instance.
- **Token** Pointer to the token used to close control connection.

#### Description
The Close() function will initiate a close request to the remote FTP server with the corresponding connection token. If this function returns EFI_SUCCESS, the control connection with the remote FTP server is closed.
Status Codes Returned

<table>
<thead>
<tr>
<th>Code</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>EFI_SUCCESS</td>
<td>The close request is successfully initiated.</td>
</tr>
</tbody>
</table>
| EFI_INVALID_PARAMETER     | One or more of the following are `TRUE`:
|                           | • This is `NULL`.                                                           |
|                           | • ConnectionToken is `NULL`.                                                |
|                           | • ConnectionToken->Event is `NULL`.                                          |
| EFI_NOT_STARTED           | The EFI FTPv4 Protocol driver has not been started.                         |
| EFI_NO_MAPPING            | When using a default address, configuration (DHCP, BOOTP, RARP, etc.) is not finished yet. |
| EFI_OUT_OF_RESOURCES      | Could not allocate enough resource to finish the operation.                |
| EFI_DEVICE_ERROR          | An unexpected system or network error occurred.                             |

**EFI_FTP4_PROTOCOL.Configure()**

**Summary**

Sets or clears the operational parameters for the FTP child driver.

**Prototype**

```c
typedef EFI_STATUS (EFIAPI *EFI_FTP4_CONFIGURE) (   
   IN EFI_FTP4_PROTOCOL *This,   
   IN EFI_FTP4_CONFIG_DATA *FtpConfigData OPTIONAL   
);
```

**Parameters**

- **This**: Pointer to the `EFI_FTP4_PROTOCOL` instance.
- **FtpConfigData**: Pointer to configuration data that will be assigned to the FTP child driver instance. If `NULL`, the FTP child driver instance is reset to startup defaults and all pending transmit and receive requests are flushed.
Related Definitions

```c
//***************************************************************
// EFI_FTP4_CONFIG_DATA
//***************************************************************
typedef struct {
  UINT8                *Username;
  UINT8                *Password;
  BOOLEAN              Active;
  BOOLEAN              UseDefaultSetting;
  EFI_IPv4_ADDRESS     StationIp;
  EFI_IPv4_ADDRESS     SubnetMask;
  EFI_IPv4_ADDRESS     GatewayIp;
  EFI_IPv4_ADDRESS     ServerIp;
  UINT16               ServerPort;
  UINT16               AltDataPort;
  UINT8                RepType;
  UINT8                FileStruct;
  UINT8                TransMode;
} EFI_FTP4_CONFIG_DATA;
```

Username  
Pointer to a ASCII string that contains user name. The caller is responsible for freeing Username after GetModeData() is called.

Password  
Pointer to a ASCII string that contains password. The caller is responsible for freeing Password after GetModeData() is called.

Active  
Set it to TRUE to initiate an active data connection. Set it to FALSE to initiate a passive data connection.

UseDefaultSetting  
Boolean value indicating if default network setting used.

StationIp  
IP address of station if UseDefaultSetting is FALSE.

SubnetMask  
Subnet mask of station if UseDefaultSetting is FALSE.

GatewayIp  
IP address of gateway if UseDefaultSetting is FALSE.

ServerIp  
IP address of FTPv4 server.

ServerPort  
FTPv4 server port number of control connection, and the default value is 21 as convention.

AltDataPort  
FTPv4 server port number of data connection. If it is zero, use (ServerPort - 1) by convention.

RepType  
A byte indicate the representation type. The right 4 bit is used for first parameter, the left 4 bit is use for second parameter

- For the first parameter, 0x0 = image, 0x1 = EBCDIC, 0x2 = ASCII, 0x3 = local
- For the second parameter, 0x0 = Non-print, 0x1 = Telnet format effectors, 0x2 = Carriage Control
• If it is a local type, the second parameter is the local byte size.
• If it is a image type, the second parameter is undefined.

FileStruct
Defines the file structure in FTP used. 0x00 = file, 0x01 =
record, 0x02 = page

TransMode
Defines the transfer mode used in FTP. 0x00 = stream, 0x01 = Block,
0x02 = Compressed

Description
The Configure() function will configure the connected FTP session with the configuration setting
specified in FtpConfigData. The configuration data can be reset by calling Configure() with
FtpConfigData set to NULL.

Status Codes Returned.

<table>
<thead>
<tr>
<th>Code</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>EFI_SUCCESS</td>
<td>The FTPv4 driver was configured successfully.</td>
</tr>
<tr>
<td>EFI_INVALID_PARAMETER</td>
<td>One or more following conditions are TRUE:</td>
</tr>
<tr>
<td></td>
<td>• This is NULL.</td>
</tr>
<tr>
<td></td>
<td>• FtpConfigData.RepType is invalid.</td>
</tr>
<tr>
<td></td>
<td>• FtpConfigData.FileStruct is invalid.</td>
</tr>
<tr>
<td></td>
<td>• FtpConfigData.TransMode is invalid.</td>
</tr>
<tr>
<td></td>
<td>• IP address in FtpConfigData is invalid.</td>
</tr>
<tr>
<td>EFI_NO_MAPPING</td>
<td>When using a default address, configuration (DHCP, BOOTP, RARP, etc.)</td>
</tr>
<tr>
<td></td>
<td>has not finished yet.</td>
</tr>
<tr>
<td>EFI_UNSUPPORTED</td>
<td>One or more of the configuration parameters are not supported by this</td>
</tr>
<tr>
<td></td>
<td>implementation.</td>
</tr>
<tr>
<td>EFI_OUT_OF_RESOURCES</td>
<td>The EFI FTPv4 Protocol driver instance data could not be allocated.</td>
</tr>
<tr>
<td>EFI_DEVICE_ERROR</td>
<td>An unexpected system or network error occurred. The EFI FTPv4 Protocol</td>
</tr>
<tr>
<td></td>
<td>driver instance is not configured.</td>
</tr>
</tbody>
</table>

**EFI_FTP4_PROTOCOL.ReadFile()**

Summary
Downloads a file from an FTPv4 server.

Prototype

typedef
EFI_STATUS
(EIFIAPI *EFI_FTP4_READ_FILE)(
    IN EFI_FTP4_PROTOCOL    *This,
    IN EFI_FTP4_COMMAND_TOKEN *Token
);

Parameters
This Pointer to the EFI_FTP4_PROTOCOL instance.
Token

Point to the token structure to provide the parameters that are used in this operation. Type `EFI_FTP4_COMMAND_TOKEN` is defined in "Related Definitions" below.

**Related Definitions**

```c
typedef struct {
    EFI_EVENT   Event;
    UINT8       *Pathname;
    UINT64      DataBufferSize;
    VOID        *DataBuffer;
    EFI_FTP4_DATA_CALLBACK DataCallback;
    VOID        *Context;
    EFI_STATUS  Status;
} EFI_FTP4_COMMAND_TOKEN;
```

**Event**

The `Event` to signal after request is finished and `Status` field is updated by the EFI FTP v4 Protocol driver. The type of `Event` must be `EVT_NOTIFY_SIGNAL`, and its Task Priority Level (TPL) must be lower than or equal to `TPL_CALLBACK`. If it is set to `NULL`, related function must wait until the function completes.

**Pathname**

Pointer to a null-terminated ASCII name string.

**DataBufferSize**

The size of data buffer in bytes.

**DataBuffer**

Pointer to the data buffer. Data downloaded from FTP server through connection is downloaded here.

**DataCallback**

Pointer to a callback function. If it is receiving function that leads to inbound data, the callback function is called when data buffer is full. Then, old data in the data buffer should be flushed and new data is stored from the beginning of data buffer. If it is a transmit function that lead to outbound data and `DataBufferSize` of `Data` in `DataBuffer` has been transmitted, this callback function is called to supply additional data to be transmitted. The size of additional data to be transmitted is indicated in `DataBufferSize`, again. If there is no data remained, `DataBufferSize` should be set to 0.

**Context**

Pointer to the parameter for `DataCallback`.

**Status**

The variable to receive the result of the completed operation.

- **EFI_SUCCESS**
  
  The FTP command is completed successfully.

- **EFI_ACCESS_DENIED**
  
  The FTP server denied the access to the requested file.
**EFI_CONNECTION_RESET**

- The connect fails because the connection is reset either by instance itself or communication peer.

**EFI_TIMEOUT**

- The connection establishment timer expired and no more specific information is available.

**EFI_NETWORK_UNREACHABLE**

- The active open fails because an ICMP network unreachable error is received.

**EFI_HOST_UNREACHABLE**

- The active open fails because an ICMP host unreachable error is received.

**EFI_PROTOCOL_UNREACHABLE**

- The active open fails because an ICMP protocol unreachable error is received.

**EFI_PORT_UNREACHABLE**

- The connection establishment timer times out and an ICMP port unreachable error is received.

**EFI_ICMP_ERROR**

- The connection establishment timer timeout and some other ICMP error is received.

**EFI_DEVICE_ERROR**

- An unexpected system or network error occurred.

**Related Definitions**

```c
//******************************************************************************
// EFI_FTP4_DATA_CALLBACK
//******************************************************************************
typedef
EFI_STATUS
(EFIAPI *EFI_FTP4_DATA_CALLBACK)(
    IN EFI_FTP4_PROTOCOL      *This,
    IN EFI_FTP4_COMMAND_TOKEN   *Token,
);
```

- **This**
  - Pointer to the **EFI_FTP4_PROTOCOL** instance.

- **Token**
  - Pointer to the token structure to provide the parameters that are used in this operation. Type **EFI_FTP4_COMMAND_TOKEN** is defined in "Related Definitions" above.

**Description**

The **ReadFile()** function is used to initialize and start an FTPv4 download process and optionally wait for completion. When the download operation completes, whether successfully or not, the **Token.Status** field is updated by the EFI FTPv4 Protocol driver and then **Token.Event** is signaled (if it is not **NULL**).
Data will be downloaded from the FTPv4 server into `Token.DataBuffer`. If the file size is larger than `Token.DataBufferSize`, `Token.DataCallback` will be called to allow for processing data and then new data will be placed at the beginning of `Token.DataBuffer`.

### Status Codes Returned

<table>
<thead>
<tr>
<th>Status Code</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>EFI_SUCCESS</td>
<td>The data file is being downloaded successfully.</td>
</tr>
<tr>
<td>EFI_INVALID_PARAMETER</td>
<td>One or more of the parameters is not valid.</td>
</tr>
<tr>
<td>• This is NULL.</td>
<td></td>
</tr>
<tr>
<td>• Token is NULL.</td>
<td></td>
</tr>
<tr>
<td>• Token.Pathname is NULL.</td>
<td></td>
</tr>
<tr>
<td>• Token.DataBuffer is NULL.</td>
<td></td>
</tr>
<tr>
<td>• Token.DataBufferSize is 0.</td>
<td></td>
</tr>
<tr>
<td>EFI_NOT_STARTED</td>
<td>The EFI FTPv4 Protocol driver has not been started.</td>
</tr>
<tr>
<td>EFI_NO_MAPPING</td>
<td>When using a default address, configuration (DHCP, BOOTP, RARP, etc.) is not finished yet.</td>
</tr>
<tr>
<td>EFI_OUT_OF_RESOURCES</td>
<td>Required system resources could not be allocated.</td>
</tr>
<tr>
<td>EFI_DEVICE_ERROR</td>
<td>An unexpected network error or system error occurred.</td>
</tr>
</tbody>
</table>

### EFI_FTP4_PROTOCOL.WriteFile()

**Summary**

Uploads a file from an FTPv4 server.

**Prototype**

```c
typedef EFI_STATUS (EFIAPI *EFI_FTP4_WRITE_FILE)(
    IN EFI_FTP4_PROTOCOL     *This,
    IN EFI_FTP4_COMMAND_TOKEN  *Token
);
```

**Parameters**

- **This** Pointer to the `EFI_FTP4_PROTOCOL` instance.
- **Token** Pointer to the token structure to provide the parameters that are used in this operation. Type `EFI_FTP4_COMMAND_TOKEN` is defined in "EFI_FTP4_READ_FILE".

**Description**

The `WriteFile()` function is used to initialize and start an FTPv4 upload process and optionally wait for completion. When the upload operation completes, whether successfully or not, the `Token.Status` field is updated by the EFI FTPv4 Protocol driver and then `Token.Event` is signaled (if it is not `NULL`).

Data to be uploaded to server is stored into `Token.DataBuffer`. `Token.DataBufferSize` is the number bytes to be transferred. If the file size is larger than `Token.DataBufferSize`, `Token.DataCallback` will be called to allow for processing data and then new data will be placed at
the beginning of Token.DataBuffer. Token.DataBufferSize is updated to reflect the actual number of bytes to be transferred. Token.DataBufferSize is set to 0 by the call back to indicate the completion of data transfer.

**Status Codes Returned**

<table>
<thead>
<tr>
<th>Code</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>EFI_SUCCESS</td>
<td>The data file is being uploaded successfully.</td>
</tr>
<tr>
<td>EFI_UNSUPPORTED</td>
<td>The operation is not supported by this implementation.</td>
</tr>
<tr>
<td>EFI_INVALID_PARAMETER</td>
<td>One or more of the parameters is not valid.</td>
</tr>
<tr>
<td></td>
<td>• This is <strong>NULL</strong>.</td>
</tr>
<tr>
<td></td>
<td>• Token is <strong>NULL</strong>.</td>
</tr>
<tr>
<td></td>
<td>• Token.Pathname is <strong>NULL</strong>.</td>
</tr>
<tr>
<td></td>
<td>• Token.DataBuffer is <strong>NULL</strong>.</td>
</tr>
<tr>
<td></td>
<td>• Token.DataBufferSize is <strong>0</strong>.</td>
</tr>
<tr>
<td>EFI_NOT_STARTED</td>
<td>The EFI FTPv4 Protocol driver has not been started.</td>
</tr>
<tr>
<td>EFI_NO_MAPPING</td>
<td>When using a default address, configuration (DHCP, BOOTP, RARP, etc.) is not finished yet.</td>
</tr>
<tr>
<td>EFI_OUT_OF_RESOURCES</td>
<td>Required system resources could not be allocated.</td>
</tr>
<tr>
<td>EFI_DEVICE_ERROR</td>
<td>An unexpected network error or system error occurred.</td>
</tr>
</tbody>
</table>

**EFI_FTP4_PROTOCOL.ReadDirectory()**

**Summary**

Download a data file "directory" from a FTPv4 server. May be unsupported in some EFI implementations.

**Prototype**

```c
typedef EFI_STATUS
  (EFIAPI *EFI_FTP4_READ_DIRECTORY) ( 
   IN EFI_FTP4_PROTOCOL *This,
   IN EFI_FTP4_COMMAND_TOKEN *Token
);
```

**Parameters**

- **This**
  - Pointer to the **EFI_FTP4_PROTOCOL** instance.
- **Token**
  - Pointer to the token structure to provide the parameters that are used in this operation. Type **EFI_FTP4_COMMAND_TOKEN** is defined in "**EFI_FTP4_READ_FILE**".

**Description**

The **ReadDirectory()** function is used to return a list of files on the FTPv4 server that logically (or operationally) related to Token.Pathname, and optionally wait for completion. When the download operation completes, whether successfully or not, the Token.Status field is updated by the EFI FTPv4 Protocol driver and then Token.Event is signaled (if it is not **NULL**).
Data will be downloaded from the FTPv4 server into `Token.DataBuffer`. If the file size is larger than `Token.DataBufferSize`, `Token.DataCallback` will be called to allow for processing data and then new data will be placed at the beginning of `Token.DataBuffer`.

### Status Codes Returned

<table>
<thead>
<tr>
<th>Code</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>EFI_SUCCESS</td>
<td>The file list information is being downloaded successfully.</td>
</tr>
<tr>
<td>EFI_UNSUPPORTED</td>
<td>The operation is not supported by this implementation.</td>
</tr>
<tr>
<td>EFI_INVALID_PARAMETER</td>
<td>One or more of the parameters is not valid.</td>
</tr>
<tr>
<td></td>
<td>• <code>This</code> is <code>NULL</code>.</td>
</tr>
<tr>
<td></td>
<td>• <code>Token</code> is <code>NULL</code>.</td>
</tr>
<tr>
<td></td>
<td>• <code>Token.DataBuffer</code> is <code>NULL</code>.</td>
</tr>
<tr>
<td></td>
<td>• <code>Token.DataBufferSize</code> is <code>0</code>.</td>
</tr>
<tr>
<td>EFI_NOT_STARTED</td>
<td>The EFI FTPv4 Protocol driver has not been started.</td>
</tr>
<tr>
<td>EFI_NO_MAPPING</td>
<td>When using a default address, configuration (DHCP, BOOTP, RARP, etc.) is</td>
</tr>
<tr>
<td></td>
<td>not finished yet.</td>
</tr>
<tr>
<td>EFI_OUT_OF_RESOURCES</td>
<td>Required system resources could not be allocated.</td>
</tr>
<tr>
<td>EFI_DEVICE_ERROR</td>
<td>An unexpected network error or system error occurred.</td>
</tr>
</tbody>
</table>

#### EFI_FTP4_PROTOCOL.Poll()

**Summary**
Polls for incoming data packets and processes outgoing data packets.

**Prototype**

```c
typedef EFI_STATUS
  (EFIAPI *EFI_FTP4_POLL) (    
    IN EFI_FTP4_PROTOCOL    *This
  );
```

**Parameters**

- `This` Pointer to the `EFI_FTP4_PROTOCOL` instance.

**Description**

The `Poll()` function can be used by network drivers and applications to increase the rate that data packets are moved between the communications device and the transmit and receive queues.

In some systems, the periodic timer event in the managed network driver may not poll the underlying communications device fast enough to transmit and/or receive all data packets without missing incoming packets or dropping outgoing packets. Drivers and applications that are experiencing packet loss should try calling the `Poll()` function more often.
Status Codes Returned

<table>
<thead>
<tr>
<th>Code</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>EFI_SUCCESS</td>
<td>Incoming or outgoing data was processed.</td>
</tr>
<tr>
<td>EFI_NOT_STARTED</td>
<td>This EFI FTPv4 Protocol instance has not been started.</td>
</tr>
<tr>
<td>EFI_INVALID_PARAMETER</td>
<td>This is NULL.</td>
</tr>
<tr>
<td>EFI_DEVICE_ERROR</td>
<td>An unexpected system or network error occurred.</td>
</tr>
<tr>
<td>EFI_TIMEOUT</td>
<td>Data was dropped out of the transmit and/or receive queue. Consider increasing the polling rate.</td>
</tr>
</tbody>
</table>

28.10 EFI TLS Protocols

28.10.1 EFI TLS Service Binding Protocol

EFI_TLS_SERVICE_BINDING_PROTOCOL

Summary
The EFI TLS Service Binding Protocol is used to locate EFI TLS Protocol drivers to create and destroy child of the driver to communicate with other host using TLS protocol.

GUID

```
#define EFI_TLS_SERVICE_BINDING_PROTOCOL_GUID 
  { \
    0x952cb795, 0xff36, 0x48cf, 0xa2, 0x49, 0x4d, 0xf4, 0x86, 0xd6, 0xab, 0x8d \n  }
```

Description
The TLS consumer need locate EFI_TLS_SERVICE_BINDING_PROTOCOL and call CreateChild() to create a new child of EFI_TLS_PROTOCOL and EFI_TLS_CONFIGURATION_PROTOCOL instance. Then use EFI_TLS_CONFIGURATION_PROTOCOL to set TLS configuration data, and use EFI_TLS_PROTOCOL to start TLS session. After use, the TLS consumer needs to call DestroyChild() to destroy it.

28.10.2 EFI TLS Protocol

EFI_TLS_PROTOCOL

Summary
This protocol provides the ability to manage TLS session.
GUID

#define EFI_TLS_PROTOCOL_GUID \
{ 0xca959f, 0x6cfa, 0x4db1, \ 
{0x95, 0xbc, 0xe4, 0x6c, 0x47, 0x51, 0x43, 0x90 }}

Protocol Interface Structure

typedef struct _EFI_TLS_PROTOCOL {
    EFI_TLS_SET_SESSION_DATA  SetSessionData;
    EFI_TLS_GET_SESSION_DATA  GetSessionData;
    EFI_TLS_BUILD_RESPONSE_PACKET  BuildResponsePacket;
    EFI_TLS_PROCESS_PACKET  ProcessPacket;
} EFI_TLS_PROTOCOL;

Parameters

SetSessionData   Set TLS session data. See the SetSessionData () function description.

GetSessionData   Get TLS session data. See the GetSessionData () function description.

BuildResponsePacket Build response packet according to TLS state machine. This function is only valid for alert, handshake and change_cipher_spec content type. See the BuildResponsePacket () function description.

ProcessPacket   Decrypt or encrypt TLS packet during session. This function is only valid after session connected and for application_data content type. See the ProcessPacket () function description.

Description

The EFI_TLS_PROTOCOL is used to create, destroy and manage TLS session. For detail of TLS, please refer to TLS related RFC.

EFI_TLS_PROTOCOL.SetSessionData ()

Summary

Set TLS session data.
Prototype

typedef
 EFI_STATUS
 (EFIAPI *EFI_TLS_SET_SESSION_DATA)(
  IN EFI_TLS_PROTOCOL *This,
  IN EFI_TLS_SESSION_DATA_TYPE DataType,
  IN VOID *Data,
  IN UINTN DataSize
);

Parameters

This         Pointer to the EFI_TLS_PROTOCOL instance.
DataType     TLS session data type. See EFI_TLS_SESSION_DATA_TYPE
Data         Pointer to session data.
DataSize     Total size of session data.

Description

The SetSessionData() function set data for a new TLS session. All session data should be set before BuildResponsePacket() invoked.

Related Definitions

//***************************************************************
// EFI_TLS_SESSION_DATA_TYPE
//***************************************************************
typedef enum {
  EfiTlsVersion,
  EfiTlsConnectionEnd,
  EfiTlsCipherList,
  EfiTlsCompressionMethod,
  EfiTlsExtensionData,
  EfiTlsVerifyMethod,
  EfiTlsSessionID,
  EfiTlsSessionState,
  EfiTlsClientRandom,
  EfiTlsServerRandom,
  EfiTlsKeyMaterial,
  EfiTlsVerifyHost,
  EfiTlsSessionDataTypeMaximum
} EFI_TLS_SESSION_DATA_TYPE;

EfiTlsVersion   TLS session Version. The corresponding Data is of type EFI_TLS_VERSION.
EfiTlsConnectionEnd TLS session as client or as server. The corresponding Data is of EFI_TLS_CONNECTION_END.
EfiTlsCipherList  A priority list of preferred algorithms for the TLS session. The corresponding Data is a list of EFI_TLS_CIPHER.

EfiTlsCompressionMethod  TLS session compression method. The corresponding Data is of type EFI_TLS_COMPRESSION.

EfiTlsExtensionData  TLS session extension data. The corresponding Data is a list of type EFI_TLS_EXTENSION.

EfiTlsVerifyMethod  TLS session verify method. The corresponding Data is of type EFI_TLS_VERIFY.

EfiTlsSessionID  TLS session data session ID. For SetSessionData(), it is TLS session ID used for session resumption. For GetSessionData(), it is the TLS session ID used for current session. The corresponding Data is of type EFI_TLS_SESSION_ID.

EfiTlsSessionState  TLS session data session state. The corresponding Data is of type EFI_TLS_SESSION_STATE.

EfiTlsClientRandom  TLS session data client random. The corresponding Data is of type EFI_TLS_RANDOM.

EfiTlsServerRandom  TLS session data server random. The corresponding Data is of type EFI_TLS_RANDOM.

EfiTlsKeyMaterial  TLS session data key material. The corresponding Data is of type EFI_TLS_MASTER_SECRET.

EfiTlsVerifyHost  TLS session hostname for validation which is used to verify whether the name within the peer certificate matches a given host name. This parameter is invalid when EfiTlsVerifyMethod is EFI_TLS_VERIFY_NONE. The corresponding Data is of type EFI_TLS_VERIFY_HOST.

//******************************************************************************
// EFI_TLS_VERSION
//******************************************************************************
typedef struct {
    UINT8 Major;
    UINT8 Minor;
} EFI_TLS_VERSION;
**Note:** The TLS version definition is from SSL3.0 to latest TLS (e.g. 1.2). SSL2.0 is obsolete and should not be used.

```c
// EFi_TLS_CONNECTION_END
typedef enum {
    EfiTlsClient,
    EfiTlsServer,
} EFi_TLS_CONNECTION_END;
```

TLS connection end is to define TLS session as client or as server.

```c
// EFi_TLS_CIPHER
typedef struct {
    UINT8 Data1;
    UINT8 Data2;
} EFi_TLS_CIPHER;
```

**Note:** The definition of EFi_TLS_CIPHER is from RFC 5246 A.4.1. Hello Messages. The value of EFi_TLS_CIPHER is from TLS Cipher Suite Registry of IANA.

```c
// EFi_TLS_COMPRESSION
typedef UINT8 EFi_TLS_COMPRESSION;
```

**Note:** The value of EFi_TLS_COMPRESSION definition is from RFC 3749.

```c
// EFi_TLS_EXTENSION
typedef struct {
    UINT16 ExtensionType;
    UINT16 Length;
    UINT8 Data[];
} EFi_TLS_EXTENSION;
```

**Note:** The definition of EFi_TLS_EXTENSION is from RFC 5246 A.4.1. Hello Messages.
typedef UINT32 EFI_TLS_VERIFY;
#define EFI_TLS_VERIFY_NONE                  0x0
#define EFI_TLS_VERIFY_PEER                  0x1
#define EFI_TLS_VERIFY_FAIL_IF_NO_PEER_CERT  0x2
#define EFI_TLS_VERIFY_CLIENT_ONCE           0x4

The consumer needs to use either EFI_TLS_VERIFY_NONE or EFI_TLS_VERIFY_PEER.
EFI_TLS_VERIFY_FAIL_IF_NO_PEER_CERT and EFI_TLS_VERIFY_CLIENT_ONCE can be ORed with EFI_TLS_VERIFY_PEER. EFI_TLS_VERIFY_FAIL_IF_NO_PEER_CERT is only meaningful in the server mode, which means the TLS session will fail if the client certificate is absent. EFI_TLS_VERIFY_CLIENT_ONCE means the TLS session only verifies the client once, and doesn’t request a certificate during re-negotiation.

typedef struct {
    EFI_TLS_VERIFY_HOST_FLAG Flags;
    CHAR8                    *HostName;
} EFI_TLS_VERIFY_HOST;

Flags       The host name validation flags. The flags arguments can be ORed.
HostName    The specified host name to be verified.

typedef UINT32 EFI_TLS_VERIFY_HOST_FLAG;
#define EFI_TLS_VERIFY_FLAG_NONE                    0x00
#define EFI_TLS_VERIFY_FLAG_ALWAYS_CHECK_SUBJECT    0x01
#define EFI_TLS_VERIFY_FLAG_NO_WILDCARDS            0x02
#define EFI_TLS_VERIFY_FLAG_NO_PARTIAL_WILDCARDS    0x04
#define EFI_TLS_VERIFY_FLAG_MULTI_LABEL_WILDCARDS   0x08
#define EFI_TLS_VERIFY_FLAG_SINGLE_LABEL_SUBDOMAINS 0x10
#define EFI_TLS_VERIFY_FLAG_NEVER_CHECK_SUBJECT     0x20

EFI_TLS_VERIFY_FLAG_NONE means no additional flags set for hostname validation. Wildcards are supported and they match only in the left-most label.
EFI_TLS_VERIFY_FLAG_ALWAYS_CHECK_SUBJECT means to always check the Subject Distinguished Name (DN) in the peer certificate even if the certificate contains Subject Alternative Name (SAN).
EFI_TLS_VERIFY_FLAG_NO_WILDCARDS means to disable the match of all wildcards.
**EFI_TLS_VERIFY_FLAG_NO_PARTIAL_WILDCARDS** means to disable the "*" as wildcard in
labels that have a prefix or suffix (e.g. "www*" or "*www").

**EFI_TLS_VERIFY_FLAG_MULTI_LABEL_WILDCARDS** allows the "*" to match more than one
labels. Otherwise, only matches a single label.

**EFI_TLS_VERIFY_FLAG_SINGLE_LABEL_SUBDOMAINS** restricts to only match direct child
sub-domains which start with ".". For example, a name of ".example.com" would match
"www.example.com" with this flag, but would not match "www.sub.example.com".

**EFI_TLS_VERIFY_FLAG_NEVER_CHECK_SUBJECT** means never check the Subject
Distinguished Name (DN) even there is no Subject Alternative Name (SAN) in the certificate.

If both **EFI_TLS_VERIFY_FLAG_ALWAYS_CHECK_SUBJECT** and
**EFI_TLS_VERIFY_FLAG_NEVER_CHECK_SUBJECT** are specified, **EFI_INVALID_PARAMETER**
will be returned. If **EFI_TLS_VERIFY_FLAG_NO_WILDCARDS** is set with
**EFI_TLS_VERIFY_FLAG_NO_PARTIAL_WILDCARDS** or
**EFI_TLS_VERIFY_FLAG_MULTI_LABEL_WILDCARDS**, **EFI_INVALID_PARAMETER** will be
returned.

```
//******************************************************************************
// EFI_TLS_RANDOM
//******************************************************************************
typedef struct {
   UINT32  GmtUnixTime;
   UINT8   RandomBytes[28];
} EFI_TLS_RANDOM;
```

**Note:** The definition of **EFI_TLS_RANDOM** is from RFC 5246 A.4.1. Hello Messages.

```
//******************************************************************************
// EFI_TLS_MASTER_SECRET
//******************************************************************************
typedef struct {
   UINT8   Data[48];
} EFI_TLS_MASTER_SECRET;
```

**Note:** The definition of **EFI_TLS_MASTER_SECRET** is from RFC 5246 8.1. Computing the Master
Secret.
//*******************************************************************************
// EFI_TLS_SESSION_ID
//*******************************************************************************
#define MAX_TLS_SESSION_ID_LENGTH 32
typedef struct {
    UINT16 Length;
    UINT8 Data[MAX_TLS_SESSION_ID_LENGTH];
} EFI_TLS_SESSION_ID;

Note: The definition of EFI_TLS_SESSION_ID is from RFC 5246 A.4.1. Hello Messages.

//*******************************************************************************
// EFI_TLS_SESSION_STATE
//*******************************************************************************
typedef enum {
    EfiTlsSessionNotStarted,
    EfiTlsSessionHandShaking,
    EfiTlsSessionDataTransferring,
    EfiTlsSessionClosing,
    EfiTlsSessionError,
    EfiTlsSessionStateMaximum
} EFI_TLS_SESSION_STATE;

The definition of EFI_TLS_SESSION_STATE is below:

When a new child of TLS protocol is created, the initial state of TLS session is EfiTlsSessionNotStarted.

The consumer can call BuildResponsePacket() with NULL to get ClientHello to start the TLS session. Then the status is EfiTlsSessionHandShaking.

During handshake, the consumer need call BuildResponsePacket() with input data from peer, then get response packet and send to peer. After handshake finish, the TLS session status becomes EfiTlsSessionDataTransferring, and consumer can use ProcessPacket() for data transferring.

Finally, if consumer wants to active close TLS session, consumer need call SetSessionData to set TLS session state to EfiTlsSessionClosing, and call BuildResponsePacket() with NULL to get CloseNotify alert message, and sent it out.

If any error happen during parsing ApplicationData content type, EFI_ABORT will be returned by ProcessPacket(), and TLS session state will become EfiTlsSessionError. Then consumer need call BuildResponsePacket() with NULL to get alert message and sent it out.
Status Codes Returned

<table>
<thead>
<tr>
<th>Status Code</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>EFI_SUCCESS</td>
<td>The TLS session data is set successfully.</td>
</tr>
<tr>
<td>EFI_INVALID_PARAMETER</td>
<td>One or more of the following conditions is TRUE:</td>
</tr>
<tr>
<td></td>
<td>• This is NULL</td>
</tr>
<tr>
<td></td>
<td>• Data is NULL</td>
</tr>
<tr>
<td></td>
<td>• DataSize is 0.</td>
</tr>
<tr>
<td>EFI_UNSUPPORTED</td>
<td>The DataType is unsupported.</td>
</tr>
<tr>
<td>EFI_ACCESS_DENIED</td>
<td>If the DataType is one of below:</td>
</tr>
<tr>
<td></td>
<td>• EfiTlsClientRandom</td>
</tr>
<tr>
<td></td>
<td>• EfiTlsServerRandom</td>
</tr>
<tr>
<td></td>
<td>• EfiTlsKeyMaterial</td>
</tr>
<tr>
<td>EFI_NOT_READY</td>
<td>Current TLS session state is NOT</td>
</tr>
<tr>
<td></td>
<td>EfiTlsSessionStateNotStarted.</td>
</tr>
<tr>
<td>EFI_OUT_OF_RESOURCES</td>
<td>Required system resources could not be allocated.</td>
</tr>
</tbody>
</table>

**EFI_TLS_PROTOCOL.GetSessionData()**

**Summary**

Get TLS session data.

**Prototype**

```c
typedef EFI_STATUS (EFIAPI *EFI_TLS_GET_SESSION_DATA)(
    IN EFI_TLS_PROTOCOL        *This,
    IN EFI_TLS_SESSION_DATA_TYPE   DataType,
    IN OUT VOID           *Data, OPTIONAL
    IN OUT UINTN           *DataSize
);
```

**Parameters**

- **This**: Pointer to the EFI_TLS_PROTOCOL instance.
- **DataType**: TLS session data type. See EFI_TLS_SESSION_DATA_TYPE
- **Data**: Pointer to session data.
- **DataSize**: Total size of session data. On input, it means the size of Data buffer. On output, it means the size of copied Data buffer if EFI_SUCCESS, and means the size of desired Data buffer if EFI_BUFFER_TOO_SMALL.

**Description**

The GetSessionData() function return the TLS session information.
Status Codes Returned

<table>
<thead>
<tr>
<th>Code</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>EFI_SUCCESS</td>
<td>The TLS session data is got successfully.</td>
</tr>
<tr>
<td>EFI_INVALID_PARAMETER</td>
<td>One or more of the following conditions is TRUE:</td>
</tr>
<tr>
<td></td>
<td>• This is NULL.</td>
</tr>
<tr>
<td></td>
<td>• DataSize is NULL.</td>
</tr>
<tr>
<td></td>
<td>• Data is NULL if *DataSize is not zero.</td>
</tr>
<tr>
<td>EFI_UNSUPPORTED</td>
<td>The DataType is unsupported.</td>
</tr>
<tr>
<td>EFI_NOT_FOUND</td>
<td>The TLS session data is not found.</td>
</tr>
<tr>
<td>EFI_NOT_READY</td>
<td>The DataType is not ready in current session state.</td>
</tr>
<tr>
<td>EFI_BUFFER_TOO_SMALL</td>
<td>The buffer is too small to hold the data.</td>
</tr>
</tbody>
</table>

EFI_TLS_PROTOCOL.BuildResponsePacket()

**Summary**

Build response packet according to TLS state machine. This function is only valid for alert, handshake and change_cipher_spec content type.

**Prototype**

```c
typedef EFI_STATUS
    (EFIAPI *EFI_TLS_BUILD_RESPONSE_PACKET)(
    IN EFI_TLS_PROTOCOL        *This,
    IN UINT8             *RequestBuffer, OPTIONAL
    IN UINTN             RequestSize, OPTIONAL
    OUT UINT8            *Buffer, OPTIONAL
    IN OUT UINTN          *BufferSize);
```

**Parameters**

- **This**
  Pointer to the EFI_TLS_PROTOCOL instance.
- **RequestBuffer**
  Pointer to the most recently received TLS packet. NULL means TLS need initiate the TLS session and response packet need to be ClientHello.
- **RequestSize**
  Packet size in bytes for the most recently received TLS packet. 0 is only valid when RequestBuffer is NULL.
- **Buffer**
  Pointer to the buffer to hold the built packet.
- **BufferSize**
  Pointer to the buffer size in bytes. On input, it is the buffer size provided by the caller. On output, it is the buffer size in fact needed to contain the packet.

**Description**

The BuildResponsePacket() function builds TLS response packet in response to the TLS request packet specified by RequestBuffer and RequestSize. If RequestBuffer is NULL and RequestSize is 0, the returned packet is a ClientHello packet. If RequestBuffer is not NULL and RequestSize is 0, the returned packet is a ClientHello packet with a zero-length record. If RequestBuffer is NULL and RequestSize is not 0, the returned packet is a ServerHello packet.

The function can return the following status code:

- EFI_SUCCESS
  The TLS session data is got successfully.
- EFI_INVALID_PARAMETER
  One or more of the following conditions is TRUE:
  • This is NULL.
  • DataSize is NULL.
  • Data is NULL if *DataSize is not zero.
- EFI_UNSUPPORTED
  The DataType is unsupported.
- EFI_NOT_FOUND
  The TLS session data is not found.
- EFI_NOT_READY
  The DataType is not ready in current session state.
- EFI_BUFFER_TOO_SMALL
  The buffer is too small to hold the data.
is 0, and TLS session status is `EfiTlsSessionNot Started`, the TLS session will be initiated and the response packet needs to be ClientHello. If `RequestBuffer` is NULL and `RequestSize` is 0, and TLS session status is `EfiTlsSessionClosing`, the TLS session will be closed and response packet needs to be CloseNotify. If `RequestBuffer` is NULL and `RequestSize` is 0, and TLS session status is `EfiTlsSessionError`, the TLS session has errors and the response packet needs to be Alert message based on error type.

**Status Codes Returned**

<table>
<thead>
<tr>
<th>Status Code</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>EFI_SUCCESS</td>
<td>The required TLS packet is built successfully.</td>
</tr>
<tr>
<td>EFI_INVALID_PARAMETER</td>
<td>One or more of the following conditions is TRUE:</td>
</tr>
<tr>
<td></td>
<td>• <code>This</code> is NULL.</td>
</tr>
<tr>
<td></td>
<td>• <code>RequestBuffer</code> is NULL but <code>RequestSize</code> is NOT 0.</td>
</tr>
<tr>
<td></td>
<td>• <code>RequestSize</code> is 0 but <code>RequestBuffer</code> is NOT NULL.</td>
</tr>
<tr>
<td></td>
<td>• <code>BufferSize</code> is NULL.</td>
</tr>
<tr>
<td></td>
<td>• <code>Buffer</code> is NULL if <code>BufferSize</code> is not zero.</td>
</tr>
<tr>
<td>EFI_BUFFER_TOO_SMALL</td>
<td><code>BufferSize</code> is too small to hold the response packet.</td>
</tr>
<tr>
<td>EFI_NOT_READY</td>
<td>Current TLS session state is NOT ready to build <code>ResponsePacket</code>.</td>
</tr>
<tr>
<td>EFI_ABORTED</td>
<td>Something wrong build response packet.</td>
</tr>
</tbody>
</table>

**EFI_TLS_PROTOCOL.ProcessPacket ()**

**Summary**

Decrypt or encrypt TLS packet during session. This function is only valid after session connected and for application_data content type.

**Prototype**

```c
typedef EFI_STATUS (EFIAPI *EFI_TLS_PROCESS_PACKET)(
    IN EFI_TLS_PROTOCOL       *This,
    IN OUT EFI_TLS_FRAGMENT_DATA **FragmentTable,
    IN UINT32            *FragmentCount,
    IN EFI_TLS_CRYPT_MODE   CryptMode
);
```

**Parameters**

- **This** Pointer to the `EFI_TLS_PROTOCOL` instance.
- **FragmentTable** Pointer to a list of fragment. The caller will take responsible to handle the original `FragmentTable` while it may be reallocated in TLS driver. If `CryptMode` is `EfiTlsEncrypt`, on input these fragments contain the TLS header and plain text TLS APP payload; on output these fragments contain the TLS header and cypher text TLS APP payload. If `CryptMode` is `EfiTlsDecrypt`, on input these fragments contain the TLS
header and cypher text TLS APP payload; on output these fragments contain the TLS header and plain text TLS APP payload.

<table>
<thead>
<tr>
<th>FragmentCount</th>
<th>Number of fragment.</th>
</tr>
</thead>
<tbody>
<tr>
<td>CryptMode</td>
<td>Crypt mode.</td>
</tr>
</tbody>
</table>

**Description**

The ProcessPacket() function process each inbound or outbound TLS APP packet.

**Related Definitions**

//******************************************************************************************
// EFI_TLS_FRAGMENT_DATA
//******************************************************************************************
typedef struct {
    UINT32     FragmentLength;
    VOID   *FragmentBuffer;
} EFI_TLS_FRAGMENT_DATA;

FragmentLength Length of data buffer in the fragment.
FragmentBuffer Pointer to the data buffer in the fragment.

//******************************************************************************************
// EFI_TLS_CRYPT_MODE
//******************************************************************************************
typedef enum {
    EfiTlsEncrypt,
    EfiTlsDecrypt,
} EFI_TLS_CRYPT_MODE;

EfiTlsEncrypt Encrypt data provided in the fragment buffers.
EfiTlsDecrypt Decrypt data provided in the fragment buffers.
Status Codes Returned

<table>
<thead>
<tr>
<th>Code</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>EFI_SUCCESS</td>
<td>The operation completed successfully.</td>
</tr>
<tr>
<td>EFI_INVALID_PARAMETER</td>
<td>One or more of the following conditions is TRUE:</td>
</tr>
<tr>
<td></td>
<td>- This is NULL.</td>
</tr>
<tr>
<td></td>
<td>- FragmentTable is NULL.</td>
</tr>
<tr>
<td></td>
<td>- FragmentCount is NULL.</td>
</tr>
<tr>
<td></td>
<td>- CryptoMode is invalid.</td>
</tr>
<tr>
<td>EFI_NOT_READY</td>
<td>Current TLS session state is NOT EfiTlsSessionDataTransferring.</td>
</tr>
<tr>
<td>EFI_ABORTED</td>
<td>Something wrong decryption the message. TLS session status will become EfiTlsSessionError. The caller need call BuildResponsePacket() to generate Error Alert message and send it out.</td>
</tr>
<tr>
<td>EFI_OUT_OF_RESOURCES</td>
<td>No enough resource to finish the operation.</td>
</tr>
</tbody>
</table>

28.10.3 EFI TLS Configuration Protocol

**EFI_TLS_CONFIGURATION_PROTOCOL**

Summary
This protocol provides a way to set and get TLS configuration.

GUID

```c
#define EFI_TLS_CONFIGURATION_PROTOCOL_GUID \
{ 0x1682fe44, 0xbd7a, 0x4407, \ 
 {0xb7, 0xc7, 0xdc, 0xa3, 0x7c, 0xa3, 0x92, 0x2d }}
```

Protocol Interface Structure

```c
typedef struct _EFI_TLS_CONFIGURATION_PROTOCOL {
  EFI_TLS_CONFIGURATION_SET_DATA    SetData;
  EFI_TLS_CONFIGURATION_GET_DATA    GetData;
} EFI_TLS_CONFIGURATION_PROTOCOL;
```

Parameters

- **SetData**
  Set TLS configuration data. See the `SetData()` function description.

- **GetData**
  Get TLS configuration data. See the `GetData()` function description.

Description

The **EFI_TLS_CONFIGURATION_PROTOCOL** is designed to provide a way to set and get TLS configuration, such as Certificate, private key file.
EFI_TLS_CONFIGURATION_PROTOCOL.SetData()

Summary
Set TLS configuration data.

Prototype

typedef

EFI_STATUS

(EIFIAPI *EFI_TLS_CONFIGURATION_SET_DATA)(

IN EFI_TLS_CONFIGURATION_PROTOCOL *This,
IN EFI_TLS_CONFIG_DATA_TYPE.DataType,
IN VOID *Data,
IN UINTN DataSize
);

Parameters

This Pointer to the EFI_TLS_CONFIGURATION_PROTOCOL instance.
DataType Configuration data type. See EFI_TLS_CONFIG_DATA_TYPE
Data Pointer to configuration data.
DataSize Total size of configuration data.

Description

The SetData() function sets TLS configuration to non-volatile storage or volatile storage.

Related Definitions

//**************************************************************
// EFI_TLS_CONFIG_DATA_TYPE
//**************************************************************
typedef enum {
 EfiTlsConfigDataTypeHostPublicCert,
 EfiTlsConfigDataTypeHostPrivateKey,
 EfiTlsConfigDataTypeCACertificate,
 EfiTlsConfigDataTypeCertRevocationList,
 EfiTlsConfigDataTypeTypeMaximum
} EFI_TLS_CONFIG_DATA_TYPE;

EfiTlsConfigDataTypeHostPublicCert
Local host configuration data: public certificate data. This data should be DER-encoded binary X.509 certificate or PEM-encoded X.509 certificate.

EfiTlsConfigDataTypeHostPrivateKey
Local host configuration data: private key data.

EfiTlsConfigDataTypeCACertificate
CA certificate to verify peer. This data should be PEM-encoded RSA or PKCS#8 private key.

EfiTlsConfigDataTypeCertRevocationList
CA-supplied Certificate Revocation List data. This data should be DER-encoded CRL data.

### Status Codes Returned

<table>
<thead>
<tr>
<th>Status Code</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>EFI_SUCCESS</td>
<td>The TLS configuration data is set successfully.</td>
</tr>
<tr>
<td>EFI_INVALID_PARAMETER</td>
<td>One or more of the following conditions is TRUE:</td>
</tr>
<tr>
<td>Data</td>
<td>• This is NULL.</td>
</tr>
<tr>
<td>DataSize</td>
<td>• Data is NULL.</td>
</tr>
<tr>
<td></td>
<td>• DataSize is 0.</td>
</tr>
<tr>
<td>EFI_UNSUPPORTED</td>
<td>The DataType is unsupported.</td>
</tr>
<tr>
<td>EFI_OUT_OF_RESOURCES</td>
<td>Required system resources could not be allocated.</td>
</tr>
</tbody>
</table>

### EFI_TLS_CONFIGURATION_PROTOCOL.GetData()

**Summary**

Get TLS configuration data.

**Prototype**

```c
typedef
    EFI_STATUS
(EFIAPI *EFI_TLS_CONFIGURATION_GET_DATA)(
    IN EFI_TLS_CONFIGURATION_PROTOCOL *This,
    IN EFI_TLS_CONFIG_DATA_TYPE     DataType,
    IN OUT VOID             *Data,
    OPTIONAL IN OUT UINTN    *DataSize
);
```

**Parameters**

- **This**
  - Pointer to the EFI_TLS_CONFIGURATION_PROTOCOL instance.
- **DataType**
  - Configuration data type. See EFI_TLS_CONFIG_DATA_TYPE
- **Data**
  - Pointer to configuration data.
- **DataSize**
  - Total size of configuration data. On input, it means the size of Data buffer. On output, it means the size of copied Data buffer if EFI_SUCCESS, and means the size of desired Data buffer if EFI_BUFFER_TOO_SMALL.

**Description**

The GetData() function gets TLS configuration.
### Status Codes Returned

<table>
<thead>
<tr>
<th>Code</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>EFI_SUCCESS</td>
<td>The TLS configuration data is got successfully.</td>
</tr>
<tr>
<td>EFI_INVALID_PARAMETER</td>
<td>One or more of the following conditions is TRUE:</td>
</tr>
<tr>
<td></td>
<td>• <strong>This</strong> is NULL.</td>
</tr>
<tr>
<td></td>
<td>• <strong>DataSize</strong> is NULL</td>
</tr>
<tr>
<td></td>
<td>• <strong>Data</strong> is NULL if <em>DataSize</em> is not zero.</td>
</tr>
<tr>
<td>EFI_UNSUPPORTED</td>
<td>The <strong>DataType</strong> is unsupported.</td>
</tr>
<tr>
<td>EFI_NOT_FOUND</td>
<td>The TLS configuration data is not found.</td>
</tr>
<tr>
<td>EFI_BUFFER_TOO_SMALL</td>
<td>The buffer is too small to hold the data.</td>
</tr>
</tbody>
</table>
29.1 ARP Protocol

This section defines the EFI Address Resolution Protocol (ARP) Protocol interface. It is split into the following two main sections:

- ARP Service Binding Protocol (ARPSBP)
- ARP Protocol (ARP)

ARP provides a generic implementation of the Address Resolution Protocol that is described in RFCs 826 and 1122. For RFCs can be found see “Links to UEFI-Related Documents” (http://uefi.org/uefi) under the heading “IETF” (RFCs 826 and 1122) for details for code of ICMP message.

EFI_ARP_SERVICE_BINDING_PROTOCOL

Summary

The ARPSBP is used to locate communication devices that are supported by an ARP driver and to create and destroy instances of the ARP child protocol driver.

The EFI Service Binding Protocol in Section 2.5.8 defines the generic Service Binding Protocol functions. This section discusses the details that are specific to the ARP.

GUID

```c
#define EFI_ARP_SERVICE_BINDING_PROTOCOL_GUID \
{0xf44c00ee,0x1f2c,0x4a00,\} \
{0xaa,0x09,0x1c,0x9f,0x3e,0x08,0x00,0xa3}
```

Description

A network application (or driver) that requires network address resolution can use one of the protocol handler services, such as `BS->LocateHandleBuffer()`, to search for devices that publish a ARPSBP GUID. Each device with a published ARPSBP GUID supports ARP and may be available for use.

After a successful call to the `EFI_ARP_SERVICE_BINDING_PROTOCOL.CreateChild()` function, the child ARP driver instance is in an unconfigured state; it is not ready to resolve addresses.

All child ARP driver instances that are created by one `EFI_ARP_SERVICE_BINDING_PROTOCOL` instance will share an ARP cache to improve efficiency.

Before a network application terminates execution, every successful call to the `EFI_ARP_SERVICE_BINDING_PROTOCOL.CreateChild()` function must be matched with a call to the `EFI_ARP_SERVICE_BINDING_PROTOCOL.DestroyChild()` function.
Note: All the network addresses that are described in EFI_ARP_PROTOCOL are stored in network byte order. Both incoming and outgoing ARP packets are also in network byte order. All other parameters that are defined in functions or data structures are stored in host byte order.

EFI_ARP_PROTOCOL

Summary
ARP is used to resolve local network protocol addresses into network hardware addresses.

GUID
#define EFI_ARP_PROTOCOL_GUID
{0xf4b427bb,0xba21,0x4f16,
{0xbc,0x4e,0x43,0xe4,0x16,0xab,0x61,0x9c}

Protocol Interface Structure
typedef struct _EFI_ARP_PROTOCOL {
    EFI_ARP_CONFIGURE Configure;
    EFI_ARP_ADD Add;
    EFI_ARP_FIND Find;
    EFI_ARP_DELETE Delete;
    EFI_ARP_FLUSH Flush;
    EFI_ARP_REQUEST Request;
    EFI_ARP_CANCEL Cancel;
} EFI_ARP_PROTOCOL;

Parameters
  Configure Adds a new station address (protocol type and network address) to the ARP cache. See the Configure() function description.
  Add Manually inserts an entry to the ARP cache for administrative purpose. See the Add() function description.
  Find Locates one or more entries in the ARP cache. See the Find() function description.
  Delete Removes an entry from the ARP cache. See the Delete() function description.
  Flush Removes all dynamic ARP cache entries of a specified protocol type. See the Flush() function description.
  Request Starts an ARP request session. See the Request() function description.
  Cancel Abort previous ARP request session. See the Cancel() function description.

Description
The EFI_ARP_PROTOCOL defines a set of generic ARP services that can be used by any network protocol driver to resolve subnet local network addresses into hardware addresses. Normally, a periodic timer event internally sends and receives packets for ARP. But in some systems where the periodic timer is not
supported, drivers and applications that are experiencing packet loss should try calling the \texttt{Poll()} function of the EFI Managed Network Protocol frequently.

\textbf{Note:} \texttt{Add()} and \texttt{Delete()} are typically used for administrative purposes, such as denying traffic to and from a specific remote machine, preventing ARP requests from coming too fast, and providing static address pairs to save time. \texttt{Find()} is also used to update an existing ARP cache entry.

\texttt{EFI_ARP_PROTOCOL.Configure()}

\textbf{Summary}
Assigns a station address (protocol type and network address) to this instance of the ARP cache.

\textbf{Prototype}
\begin{verbatim}
typedef EFI_STATUS (EFIAPI *EFI_ARP_CONFIGURE) ( 
    IN EFI_ARP_PROTOCOL *This, 
    IN EFI_ARP_CONFIG_DATA *ConfigData OPTIONAL );
\end{verbatim}

\textbf{Parameters}
- \texttt{This} A pointer to the \texttt{EFI_ARP_PROTOCOL} instance.
- \texttt{ConfigData} A pointer to the \texttt{EFI_ARP_CONFIG_DATA} structure. Type \texttt{EFI_ARP_CONFIG_DATA} is defined in “Related Definitions” below.

\textbf{Description}
The \texttt{Configure()} function is used to assign a station address to the ARP cache for this instance of the ARP driver. Each ARP instance has one station address. The \texttt{EFI_ARP_PROTOCOL} driver will respond to ARP requests that match this registered station address. A call to \texttt{Configure()} with the \texttt{ConfigData} field set to \texttt{NULL} will reset this ARP instance.

Once a protocol type and station address have been assigned to this ARP instance, all the following ARP functions will use this information. Attempting to change the protocol type or station address to a configured ARP instance will result in errors.
Related Definitions

```c
typedef struct {  
UINT16   SwAddressType;
UINT8    SwAddressLength;
VOID     *StationAddress;
UINT32   EntryTimeOut;
UINT32   RetryCount;
UINT32   RetryTimeOut;
} EFI_ARP_CONFIG_DATA;
```

- **SwAddressType**: 16-bit protocol type number in host byte order. For more information see “Links to UEFI-Related Documents” ([http://uefi.org/uefi](http://uefi.org/uefi)) under the heading “16-bit protocol type numbers”.

- **SwAddressLength**: Length in bytes of the station’s protocol address to register.

- **StationAddress**: Pointer to the first byte of the protocol address to register. For example, if `SwAddressType` is 0x0800 (IP), then `StationAddress` points to the first byte of this station’s IP address stored in network byte order.

- **EntryTimeOut**: The timeout value in 100-ns units that is associated with each new dynamic ARP cache entry. If it is set to zero, the value is implementation-specific.

- **RetryCount**: The number of retries before a MAC address is resolved. If it is set to zero, the value is implementation-specific.

- **RetryTimeOut**: The timeout value in 100-ns units that is used to wait for the ARP reply packet or the timeout value between two retries. Set to zero to use implementation-specific value.
Status Codes Returned

<table>
<thead>
<tr>
<th>Code</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>EFI_SUCCESS</td>
<td>The new station address was successfully registered.</td>
</tr>
<tr>
<td>EFI_INVALID_PARAMETER</td>
<td>One or more of the following conditions is <strong>TRUE</strong>:</td>
</tr>
<tr>
<td></td>
<td>• This is <strong>NULL</strong>.</td>
</tr>
<tr>
<td></td>
<td>• SwAddressLength is zero when ConfigData is not <strong>NULL</strong>.</td>
</tr>
<tr>
<td></td>
<td>• StationAddress is <strong>NULL</strong> when ConfigData is not <strong>NULL</strong>.</td>
</tr>
<tr>
<td>EFI_ACCESS_DENIED</td>
<td>The SwAddressType, SwAddressLength, or StationAddress is different from the one that is already registered.</td>
</tr>
<tr>
<td>EFI_OUT_OF_RESOURCES</td>
<td>Storage for the new StationAddress could not be allocated.</td>
</tr>
</tbody>
</table>

**EFI_ARP_PROTOCOL.Add()**

**Summary**

Inserts an entry to the ARP cache.

**Prototype**

```c
typedef
  EFI_STATUS
  (EFIAPI *EFI_ARP_ADD) (    
  IN EFI_ARP_PROTOCOL *This,  
  IN BOOLEAN DenyFlag,       
  IN VOID *TargetSwAddress OPTIONAL, 
  IN VOID *TargetHwAddress OPTIONAL, 
  IN UINT32 TimeoutValue,    
  IN BOOLEAN Overwrite    
);    
```

**Parameters**

- **This**
  A pointer to the **EFI_ARP_PROTOCOL** instance.

- **DenyFlag**
  Set to **TRUE** if this entry is a “deny” entry. Set to **FALSE** if this entry is a “normal” entry.

- **TargetSwAddress**
  Pointer to a protocol address to add (or deny). May be set to **NULL** if DenyFlag is **TRUE**.

- **TargetHwAddress**
  Pointer to a hardware address to add (or deny). May be set to **NULL** if DenyFlag is **TRUE**.

- **TimeoutValue**
  Time in 100-ns units that this entry will remain in the ARP cache. A value of zero means that the entry is permanent. A nonzero value will override the one given by **Configure()** if the entry to be added is dynamic entry.

- **Overwrite**
  If **TRUE**, the matching cache entry will be overwritten with the supplied parameters. If **FALSE**, **EFI_ACCESS_DENIED** is returned if the corresponding cache entry already exists.
Description
The Add() function is used to insert entries into the ARP cache.

ARP cache entries are typically inserted and updated by network protocol drivers as network traffic is processed. Most ARP cache entries will time out and be deleted if the network traffic stops. ARP cache entries that were inserted by the Add() function may be static (will not time out) or dynamic (will time out).

Default ARP cache timeout values are not covered in most network protocol specifications (although RFC 1122 comes pretty close) and will only be discussed in general in this specification. The timeout values that are used in the EFI Sample Implementation should be used only as a guideline. Final product implementations of the EFI network stack should be tuned for their expected network environments.

The Add() function can insert the following two types of entries into the ARP cache:

- “Normal” entries
- “Deny” entries

“Normal” entries must have both a TargetSwAddress and TargetHwAddress and are used to resolve network protocol addresses into network hardware addresses. Entries are keyed by TargetSwAddress. Each TargetSwAddress can have only one TargetHwAddress. A TargetHwAddress may be referenced by multiple TargetSwAddress entries.

“Deny” entries may have a TargetSwAddress or a TargetHwAddress, but not both. These entries tell the ARP driver to ignore any traffic to and from (and to) these addresses. If a request comes in from an address that is being denied, then the request is ignored.

If a normal entry to be added matches a deny entry of this driver, Overwrite decides whether to remove the matching deny entry. On the other hand, an existing normal entry can be removed based on the value of Overwrite if a deny entry to be added matches the existing normal entry. Two entries are matched only when they have the same addresses or when one of the normal entry addresses is the same as the address of a deny entry.

Status Codes Returned

<table>
<thead>
<tr>
<th>Status Code</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>EFI_SUCCESS</td>
<td>The entry has been added or updated.</td>
</tr>
<tr>
<td>EFI_INVALID_PARAMETER</td>
<td>One or more of the following conditions is TRUE:</td>
</tr>
<tr>
<td></td>
<td>This is NULL. DenyFlag is FALSE and TargetHwAddress is NULL.</td>
</tr>
<tr>
<td></td>
<td>DenyFlag is FALSE and TargetSwAddress is NULL.</td>
</tr>
<tr>
<td></td>
<td>TargetHwAddress is NULL and TargetSwAddress is NULL.</td>
</tr>
<tr>
<td></td>
<td>Both TargetSwAddress and TargetHwAddress are not NULL when DenyFlag is TRUE.</td>
</tr>
<tr>
<td>EFI_OUT_OF_RESOURCES</td>
<td>The new ARP cache entry could not be allocated.</td>
</tr>
<tr>
<td>EFI_ACCESS_DENIED</td>
<td>The ARP cache entry already exists and Overwrite is not TRUE.</td>
</tr>
<tr>
<td>EFI_NOT_STARTED</td>
<td>The ARP driver instance has not been configured.</td>
</tr>
</tbody>
</table>
EFI_ARP_PROTOCOL.Find()

Summary
Locates one or more entries in the ARP cache.

Prototype

typedef
 EFI_STATUS
 (EFIAPI *EFI_ARP_FIND) (  
  IN EFI_ARP_PROTOCOL *This,
  IN BOOLEAN BySwAddress,
  IN VOID *AddressBuffer OPTIONAL,
  OUT UINT32 *EntryLength OPTIONAL,
  OUT UINT32 *EntryCount OPTIONAL,
  OUT EFI_ARP_FIND_DATA **Entries OPTIONAL,
  IN BOOLEAN Refresh
 );

Parameters

  This A pointer to the EFI_ARP_PROTOCOL instance.
  BySwAddress Set to TRUE to look for matching software protocol addresses.
                Set to FALSE to look for matching hardware protocol addresses.
  AddressBuffer Pointer to address buffer. Set to NULL to match all addresses.
  EntryLength The size of an entry in the entries buffer. To keep the
               EFI_ARP_FIND_DATA structure properly aligned, this field may be
               longer than sizeof(EFI_ARP_FIND_DATA) plus the length of the
               software and hardware addresses.
  EntryCount The number of ARP cache entries that are found by the specified
               criteria.
  Entries Pointer to the buffer that will receive the ARP cache entries. Type
           EFI_ARP_FIND_DATA is defined in “Related Definitions” below.
  Refresh Set to TRUE to refresh the timeout value of the matching ARP cache
           entry.

Description

The Find() function searches the ARP cache for matching entries and allocates a buffer into which those
entries are copied. The first part of the allocated buffer is EFI_ARP_FIND_DATA, following which are
protocol address pairs and hardware address pairs.

When finding a specific protocol address (BySwAddress is TRUE and AddressBuffer is not NULL), the
ARP cache timeout for the found entry is reset if Refresh is set to TRUE. If the found ARP cache entry is a
permanent entry, it is not affected by Refresh.
Related Definitions

```c
typedef struct {
    UINT32 Size;
    BOOLEAN DenyFlag;
    BOOLEAN StaticFlag;
    UINT16 HwAddressType;
    UINT16 SwAddressType;
    UINT8 HwAddressLength;
    UINT8 SwAddressLength;
} EFI_ARP_FIND_DATA;
```

- **Size**: Length in bytes of this entry.
- **DenyFlag**: Set to `TRUE` if this entry is a “deny” entry. Set to `FALSE` if this entry is a “normal” entry.
- **StaticFlag**: Set to `TRUE` if this entry will not time out. Set to `FALSE` if this entry will time out.
- **HwAddressType**: 16-bit ARP hardware identifier number.
- **SwAddressType**: 16-bit protocol type number.
- **HwAddressLength**: Length of the hardware address.
- **SwAddressLength**: Length of the protocol address.

### Status Codes Returned

<table>
<thead>
<tr>
<th>Code</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>EFI_SUCCESS</td>
<td>The requested ARP cache entries were copied into the buffer.</td>
</tr>
</tbody>
</table>
| EFI_INVALID_PARAMETER       | One or more of the following conditions is `TRUE`:
|                             | • This is `NULL`.                                            |
|                             | • Both `EntryCount` and `EntryLength` are `NULL`, when `Refresh` is `FALSE`. |
| EFI_NOT_FOUND               | No matching entries were found.                             |
| EFI_NOT_STARTED             | The ARP driver instance has not been configured.            |

### EFI_ARP_PROTOCOL.Delete()

**Summary**

Removes entries from the ARP cache.
Prototype

typedef

EFI_STATUS

(EIFI_API *EFI_ARP_DELETE) (  
  IN EFI_ARP_PROTOCOL *This,
  IN BOOLEAN BySwAddress,
  IN VOID *AddressBuffer OPTIONAL
 );

Parameters

This A pointer to the EFI_ARP_PROTOCOL instance.
BySwAddress Set to TRUE to delete matching protocol addresses.  
Set to FALSE to delete matching hardware addresses.
AddressBuffer Pointer to the address buffer that is used as a key to look for the 
  cache entry. Set to NULL to delete all entries.

Description

The Delete() function removes specified ARP cache entries.

Status Codes Returned

<table>
<thead>
<tr>
<th>Code</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>EFI_SUCCESS</td>
<td>The entry was removed from the ARP cache.</td>
</tr>
<tr>
<td>EFI_INVALID_PARAMETER</td>
<td>This is NULL.</td>
</tr>
<tr>
<td>EFI_NOT_FOUND</td>
<td>The specified deletion key was not found.</td>
</tr>
<tr>
<td>EFI_NOT_STARTED</td>
<td>The ARP driver instance has not been configured.</td>
</tr>
</tbody>
</table>

EFI_ARP_PROTOCOL.Flush()

Summary

Removes all dynamic ARP cache entries that were added by this interface.

Prototype

typedef

EFI_STATUS

(EIFI_API *EFI_ARP_FLUSH) (  
  IN EFI_ARP_PROTOCOL *This
 );

Parameters

This A pointer to the EFI_ARP_PROTOCOL instance.

Description

The Flush() function deletes all dynamic entries from the ARP cache that match the specified software protocol type.
Status Codes Returned

<table>
<thead>
<tr>
<th>Code</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>EFI_SUCCESS</td>
<td>The cache has been flushed.</td>
</tr>
<tr>
<td>EFI_INVALID_PARAMETER</td>
<td>This is NULL.</td>
</tr>
<tr>
<td>EFI_NOT_FOUND</td>
<td>There are no matching dynamic cache entries.</td>
</tr>
<tr>
<td>EFI_NOT_STARTED</td>
<td>The ARP driver instance has not been configured.</td>
</tr>
</tbody>
</table>

### EFI_ARP_PROTOCOL.Request()

**Summary**

Starts an ARP request session.

**Prototype**

```c
typedef EFI_STATUS (EFIAPIC *EFI_ARP_REQUEST) ( 
  IN EFI_ARP_PROTOCOL *This, 
  IN VOID *TargetSwAddress OPTIONAL, 
  IN EFI_EVENT ResolvedEvent OPTIONAL, 
  OUT VOID *TargetHwAddress
);
```

**Parameters**

- **This**
  
  A pointer to the `EFI_ARP_PROTOCOL` instance.

- **TargetSwAddress**
  
  Pointer to the protocol address to resolve.

- **ResolvedEvent**
  
  Pointer to the event that will be signaled when the address is resolved or some error occurs.

- **TargetHwAddress**
  
  Pointer to the buffer for the resolved hardware address in network byte order. The buffer must be large enough to hold the resulting hardware address. `TargetHwAddress` must not be NULL.

**Description**

The `Request()` function tries to resolve the `TargetSwAddress` and optionally returns a `TargetHwAddress` if it already exists in the ARP cache.

If the registered `SwAddressType` (see `EFI_ARP_PROTOCOL.Add()`) is IPv4 or IPv6 and the `TargetSwAddress` is a multicast address, then the `TargetSwAddress` is resolved using the underlying `EFI_MANAGED_NETWORK_PROTOCOL.McastIpToMac()` function.

If the `TargetSwAddress` is NULL, then the network interface hardware broadcast address is returned immediately in `TargetHwAddress`.

If the `ResolvedEvent` is not NULL and the address to be resolved is not in the ARP cache, then the event will be signaled when the address request completes and the requested hardware address is returned in the `TargetHwAddress`. If the timeout expires and the retry count is exceeded or an unexpected error occurs, the event will be signaled to notify the caller, which should check the `TargetHwAddress` to see if the requested hardware address is available. If it is not available, the `TargetHwAddress` is filled by zero.
If the address to be resolved is already in the ARP cache and resolved, then the event will be signaled immediately if it is not **NULL**, and the requested hardware address is also returned in `TargetHwAddress`.

### Status Codes Returned

<table>
<thead>
<tr>
<th>Status Code</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>EFI_SUCCESS</td>
<td>The data was copied from the ARP cache into the <code>TargetHwAddress</code> buffer.</td>
</tr>
<tr>
<td>EFI_INVALID_PARAMETER</td>
<td>One or more of the following conditions is <strong>TRUE</strong>: This is <strong>NULL</strong> TargetHwAddress is <strong>NULL</strong></td>
</tr>
<tr>
<td>EFI_ACCESS_DENIED</td>
<td>The requested address is not present in the normal ARP cache but is present in the deny address list. Outgoing traffic to that address is forbidden.</td>
</tr>
<tr>
<td>EFI_NOT_STARTED</td>
<td>The ARP driver instance has not been configured.</td>
</tr>
<tr>
<td>EFI_NOT_READY</td>
<td>The request has been started and is not finished.</td>
</tr>
<tr>
<td>EFI_UNSUPPORTED</td>
<td>The requested conversion is not supported in this implementation or configuration.</td>
</tr>
</tbody>
</table>

### EFI_ARP_PROTOCOL.Cancel()

#### Summary
Cancels an ARP request session.

#### Prototype

```c
typedef
    EFI_STATUS
    (EFIAPI *EFI_ARP_CANCEL) (  
    IN EFI_ARP_PROTOCOL *This,
    IN VOID *TargetSwAddress OPTIONAL,
    IN EFI_EVENT ResolvedEvent OPTIONAL
    );
```

#### Parameters

- **This** A pointer to the **EFI_ARP_PROTOCOL** instance.
- **TargetSwAddress** Pointer to the protocol address in previous request session.
- **ResolvedEvent** Pointer to the event that is used as the notification event in previous request session.

#### Description
The `Cancel()` function aborts the previous ARP request (identified by `This`, `TargetSwAddress` and `ResolvedEvent`) that is issued by `EFI_ARP_PROTOCOL.Request()`. If the request is in the internal ARP request queue, the request is aborted immediately and its `ResolvedEvent` is signaled. Only an asynchronous address request needs to be canceled. If `TargetSwAddress` and `ResolvedEvent` are both **NULL**, all the pending asynchronous requests that have been issued by `This` instance will be cancelled and their corresponding events will be signaled.
29.2 EFI DHCPv4 Protocol

This section provides a detailed description of the `EFI_DHCP4_PROTOCOL` and the `EFI_DHCP4_SERVICE_BINDING_PROTOCOL`. The EFI DHCPv4 Protocol is used to collect configuration information for the EFI IPv4 Protocol drivers and to provide DHCPv4 server and PXE boot server discovery services.

**EFI_DHCP4_SERVICE_BINDING_PROTOCOL**

*Summary*

The EFI DHCPv4 Service Binding Protocol is used to locate communication devices that are supported by an EFI DHCPv4 Protocol driver and to create and destroy EFI DHCPv4 Protocol child driver instances that can use the underlying communications device.

**GUID**

```c
#define EFI_DHCP4_SERVICE_BINDING_PROTOCOL_GUID \
{0x9d9a39d8,0xbd42,0x4a73,\ 
{0xa4,0xd5,0x8e,0xe9,0x4b,0xe1,0x13,0x80}}
```

*Description*

A network application or driver that requires basic DHCPv4 services can use one of the protocol handler services, such as `BS->LocateHandleBuffer()`, to search for devices that publish an EFI DHCPv4 Service Binding Protocol GUID. Each device with a published EFI DHCPv4 Service Binding Protocol GUID supports the EFI DHCPv4 Protocol and may be available for use.

After a successful call to the `EFI_DHCP4_SERVICE_BINDING_PROTOCOL.CreateChild()` function, the newly created EFI DHCPv4 Protocol child driver instance is ready to be used by a network application or driver.

Before a network application or driver terminates execution, every successful call to the `EFI_DHCP4_SERVICE_BINDING_PROTOCOL.CreateChild()` function must be matched with a call to the `EFI_DHCP4_SERVICE_BINDING_PROTOCOL.DestroyChild()` function.

---

### Status Codes Returned

<table>
<thead>
<tr>
<th>Code</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>EFI_SUCCESS</td>
<td>The pending request session(s) is/are aborted and corresponding event(s) is/are signaled.</td>
</tr>
<tr>
<td>EFI_INVALID_PARAMETER</td>
<td>One or more of the following conditions is <strong>TRUE</strong>:</td>
</tr>
<tr>
<td></td>
<td>• This is <strong>NULL</strong></td>
</tr>
<tr>
<td></td>
<td>• TargetSwAddress is <strong>not NULL</strong> and ResolvedEvent is <strong>NULL</strong></td>
</tr>
<tr>
<td></td>
<td>• TargetSwAddress is <strong>NULL</strong> and ResolvedEvent is <strong>not NULL</strong></td>
</tr>
<tr>
<td>EFI_NOT_STARTED</td>
<td>The ARP driver instance has not been configured.</td>
</tr>
<tr>
<td>EFI_NOT_FOUND</td>
<td>The request is not issued by <code>EFI_ARP_PROTOCOL.Request()</code>.</td>
</tr>
</tbody>
</table>

EFI_SUCCESS

The pending request session(s) is/are aborted and corresponding event(s) is/are signaled.

EFI_INVALID_PARAMETER

One or more of the following conditions is **TRUE**:

- **This is NULL**
- TargetSwAddress is **not NULL** and ResolvedEvent is **NULL**
- TargetSwAddress is **NULL** and ResolvedEvent is **not NULL**

EFI_NOT_STARTED

The ARP driver instance has not been configured.

EFI_NOT_FOUND

The request is not issued by `EFI_ARP_PROTOCOL.Request()`.
**EFI_DHCP4_PROTOCOL**

**Summary**
This protocol is used to collect configuration information for the EFI IPv4 Protocol drivers and to provide DHCPv4 server and PXE boot server discovery services.

**GUID**
```
#define EFI_DHCP4_PROTOCOL_GUID \ 
{0x8a219718,0x4ef5,0x4761,\ 
{0x91,0xc8,0xc0,0xf0,0x4b,0xda,0x9e,0x56}}
```

**Protocol Interface Structure**
```c
typedef struct _EFI_DHCP4_PROTOCOL {
    EFI_DHCP4_GET_MODE_DATA  GetModeData;
    EFI_DHCP4_CONFIGURE      Configure;
    EFI_DHCP4_START          Start;
    EFI_DHCP4_RENEW_REBIND   RenewRebind;
    EFI_DHCP4_RELEASE        Release;
    EFI_DHCP4_STOP           Stop;
    EFI_DHCP4_BUILD          Build;
    EFI_DHCP4_TRANSMIT_RECEIVE TransmitReceive;
    EFI_DHCP4PARSE           Parse;
} EFI_DHCP4_PROTOCOL;
```

**Parameters**
- **GetModeData**
  Gets the EFI DHCPv4 Protocol driver status and operational data. See the `GetModeData()` function description.
- **Configure**
  Initializes, changes, or resets operational settings for the EFI DHCPv4 Protocol driver. See the `Configure()` function description.
- **Start**
  Starts the DHCP configuration process. See the `Start()` function description.
- **RenewRebind**
  Tries to manually extend the lease time by sending a request packet. See the `RenewRebind()` function description.
- **Release**
  Releases the current configuration and returns the EFI DHCPv4 Protocol driver to the initial state. See the `Release()` function description.
- **Stop**
  Stops the DHCP configuration process no matter what state the driver is in. After being stopped, this driver will not automatically communicate with the DHCP server. See the `Stop()` function description.
- **Build**
  Puts together a DHCP or PXE packet. See the `Build()` function description.
- **TransmitReceive**
  Transmits a DHCP or PXE packet and waits for response packets. See the `TransmitReceive()` function description.
- **Parse**
  Parses the packed DHCP or PXE option data. See the `Parse()` function description.
Description
The EFI_DHCP4_PROTOCOL is used to collect configuration information for the EFI IPv4 Protocol driver and provide DHCP server and PXE boot server discovery services.

Byte Order Note
All the IPv4 addresses that are described in EFI_DHCP4_PROTOCOL are stored in network byte order. Both incoming and outgoing DHCP packets are also in network byte order. All other parameters that are defined in functions or data structures are stored in host byte order.

EFI_DHCP4_PROTOCOL.GetModeData()

Summary
Returns the current operating mode and cached data packet for the EFI DHCPv4 Protocol driver.

Prototype
typedef
    EFI_STATUS
    (EFFIAPI *EFI_DHCP4_GET_MODE_DATA)(
    IN EFI_DHCP4_PROTOCOL *This,
    OUT EFI_DHCP4_MODE_DATA *Dhcp4ModeData
    );

Parameters
This Pointer to the EFI_DHCP4_PROTOCOL instance.
Dhcp4ModeData Pointer to storage for the EFI_DHCP4_MODE_DATA structure. Type EFI_DHCP4_MODE_DATA is defined in “Related Definitions” below.

Description
The GetModeData() function returns the current operating mode and cached data packet for the EFI DHCPv4 Protocol driver.

Related Definitions
//************************************************************************************
// EFI_DHCP4_MODE_DATA
//************************************************************************************
typedef struct {
    EFI_DHCP4_STATE State;
    EFI_DHCP4_CONFIG_DATA ConfigData;
    EFI_IPv4_ADDRESS ClientAddress;
    EFI_MAC_ADDRESS ClientMacAddress;
    EFI_IPv4_ADDRESS ServerAddress;
    EFI_IPv4_ADDRESS RouterAddress;
    EFI_IPv4_ADDRESS SubnetMask;
    UINT32 LeaseTime;
    EFI_DHCP4_PACKET *ReplyPacket;
} EFI_DHCP4_MODE_DATA;
State

The EFI DHCPv4 Protocol driver operating state. Type EFI_DHCP4_STATE is defined below.

ConfigData

The configuration data of the current EFI DHCPv4 Protocol driver instance. Type EFI_DHCP4_CONFIG_DATA is defined in EFI_DHCP4_PROTOCOL.Confirm().

ClientAddress

The client IP address that was acquired from the DHCP server. If it is zero, the DHCP acquisition has not completed yet and the following fields in this structure are undefined.

ClientMacAddress

The local hardware address.

ServerAddress

The server IP address that is providing the DHCP service to this client.

RouterAddress

The router IP address that was acquired from the DHCP server. May be zero if the server does not offer this address.

SubnetMask

The subnet mask of the connected network that was acquired from the DHCP server.

LeaseTime

The lease time (in 1-second units) of the configured IP address. The value 0xFFFFFFFF means that the lease time is infinite. A default lease of 7 days is used if the DHCP server does not provide a value.

ReplyPacket

The cached latest DHCPACK or DHCPNAK or BOOTP REPLY packet. May be NULL if no packet is cached.

The EFI_DHCP4_MODE_DATA structure describes the operational data of the current DHCP procedure.

```c
typedef enum {
    Dhcp4Stopped      = 0x0,
    Dhcp4Init         = 0x1,
    Dhcp4Selecting    = 0x2,
    Dhcp4Requesting   = 0x3,
    Dhcp4Bound        = 0x4
    Dhcp4Renewing     = 0x5,
    Dhcp4Rebinding    = 0x6,
    Dhcp4InitReboot   = 0x7,
    Dhcp4Rebooting    = 0x8
} EFI_DHCP4_STATE;
```

Table 29-1 describes the fields in the above enumeration.
Table 29-1 DHCP Enumerations

<table>
<thead>
<tr>
<th>Field</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Dhcp4Stopped</td>
<td>The EFI DHCPv4 Protocol driver is stopped and \texttt{EFI_DHCP4_PROTOCOL.Configure()} needs to be called. The rest of the \texttt{EFI_DHCP4_MODE_DATA} structure is undefined in this state.</td>
</tr>
<tr>
<td>Dhcp4Init</td>
<td>The EFI DHCPv4 Protocol driver is inactive and \texttt{EFI_DHCP4_PROTOCOL.Start()} needs to be called. The rest of the \texttt{EFI_DHCP4_MODE_DATA} structure is undefined in this state.</td>
</tr>
<tr>
<td>Dhcp4Selecting</td>
<td>The EFI DHCPv4 Protocol driver is collecting DHCP offer packets from DHCP servers. The rest of the \texttt{EFI_DHCP4_MODE_DATA} structure is undefined in this state.</td>
</tr>
<tr>
<td>Dhcp4Requesting</td>
<td>The EFI DHCPv4 Protocol driver has sent the request to the DHCP server and is waiting for a response. The rest of the \texttt{EFI_DHCP4_MODE_DATA} structure is undefined in this state.</td>
</tr>
<tr>
<td>Dhcp4Bound</td>
<td>The DHCP configuration has completed. All of the fields in the \texttt{EFI_DHCP4_MODE_DATA} structure are defined.</td>
</tr>
<tr>
<td>Dhcp4Renewing</td>
<td>The DHCP configuration is being renewed and another request has been sent out, but it has not received a response from the server yet. All of the fields in the \texttt{EFI_DHCP4_MODE_DATA} structure are available but may change soon.</td>
</tr>
<tr>
<td>Dhcp4Rebinding</td>
<td>The DHCP configuration has timed out and the EFI DHCPv4 Protocol driver is trying to extend the lease time. The rest of the \texttt{EFI_DHCP4_MODE} structure is undefined in this state.</td>
</tr>
<tr>
<td>Dhcp4InitReboot</td>
<td>The EFI DHCPv4 Protocol driver is initialized with a previously allocated or known IP address. \texttt{EFI_DHCP4_PROTOCOL.Start()} needs to be called to start the configuration process. The rest of the \texttt{EFI_DHCP4_MODE_DATA} structure is undefined in this state.</td>
</tr>
<tr>
<td>Dhcp4Rebooting</td>
<td>The EFI DHCPv4 Protocol driver is seeking to reuse the previously allocated IP address by sending a request to the DHCP server. The rest of the \texttt{EFI_DHCP4_MODE_DATA} structure is undefined in this state.</td>
</tr>
</tbody>
</table>

\texttt{EFI\_DHCP4\_STATE} defines the DHCP operational states that are described in RFC 2131, which can be obtained at “Links to UEFI-Related Documents” (http://uefi.org/uefi) under the heading “RFC 2131”.

A variable number of EFI DHCPv4 Protocol driver instances can coexist but they share the same state machine. More precisely, each communication device has a separate DHCP state machine if there are multiple communication devices. Each EFI DHCPv4 Protocol driver instance that is created by the same EFI DHCPv4 Service Binding Protocol driver instance shares the same state machine. In this document, when we refer to the state of EFI DHCPv4 Protocol driver, we actually refer to the state of the communication device from which the current EFI DHCPv4 Protocol Driver instance is created.
# EFI_DHCP4 PACKET

```c
#pragma pack(1)
typedef struct {
  UINT32 Size;
  UINT32 Length;
  struct{
    EFI_DHCP4_HEADER Header;
    UINT32 Magik;
    UINT8 Option[1];
  } Dhcp4;
} EFI_DHCP4_PACKET;
#pragma pack()
```

**Size**
Size of the **EFI_DHCP4_PACKET** buffer.

**Length**
Length of the **EFI_DHCP4_PACKET** from the first byte of the **Header** field to the last byte of the **Option[]** field.

**Header**
DHCP packet header.

**Magik**
DHCP magik cookie in network byte order.

**Option**
Start of the DHCP packed option data.

**EFI_DHCP4_PACKET** defines the format of DHCPv4 packets. See RFC 2131 for more information.

## Status Codes Returned

<table>
<thead>
<tr>
<th>Code</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>EFI_SUCCESS</td>
<td>The mode data was returned.</td>
</tr>
<tr>
<td>EFI_INVALID_PARAMETER</td>
<td>This is <strong>NULL</strong>.</td>
</tr>
</tbody>
</table>

## EFI_DHCP4_PROTOCOL.Configure()

**Summary**
Initializes, changes, or resets the operational settings for the EFI DHCPv4 Protocol driver.

**Prototype**

```c
typedef
  EFI_STATUS
  (EFI_API *EFI_DHCP4_CONFIGURE) (
    IN EFI_DHCP4_PROTOCOL *This,
    IN EFI_DHCP4_CONFIG_DATA *Dhcp4CfgData OPTIONAL
  );
```

**Parameters**

*This* Pointer to the **EFI_DHCP4_PROTOCOL** instance.
Dhcp4CfgData

Pointer to the EFI_DHCP4_CONFIG_DATA Type.

EFI_DHCP4_CONFIG_DATA is defined in “Related Definitions” below.

Description

The Configure() function is used to initialize, change, or reset the operational settings of the EFI DHCPv4 Protocol driver for the communication device on which the EFI DHCPv4 Service Binding Protocol is installed. This function can be successfully called only if both of the following are true:

- This instance of the EFI DHCPv4 Protocol driver is in the Dhcp4Stopped, Dhcp4Init, Dhcp4InitReboot, or Dhcp4Bound states.
- No other EFI DHCPv4 Protocol driver instance that is controlled by this EFI DHCPv4 Service Binding Protocol driver instance has configured this EFI DHCPv4 Protocol driver.

When this driver is in the Dhcp4Stopped state, it can transfer into one of the following two possible initial states:

- Dhcp4Init
- Dhcp4InitReboot

The driver can transfer into these states by calling Configure() with a non-NULL Dhcp4CfgData. The driver will transfer into the appropriate state based on the supplied client network address in the ClientAddress parameter and DHCP options in the OptionList parameter as described in RFC 2131.

When Configure() is called successfully while Dhcp4CfgData is set to NULL, the default configuring data will be reset in the EFI DHCPv4 Protocol driver and the state of the EFI DHCPv4 Protocol driver will not be changed. If one instance wants to make it possible for another instance to configure the EFI DHCPv4 Protocol driver, it must call this function with Dhcp4CfgData set to NULL.

Related Definitions

//**************************************************************
// EFI_DHCP4_CONFIG_DATA
//**************************************************************

typedef struct {
    UINT32 DiscoverTryCount;
    UINT32 *DiscoverTimeout;
    UINT32 RequestTryCount;
    UINT32 *RequestTimeout;
    EFI_IPv4_ADDRESS ClientAddress;
    EFI_DHCP4_CALLBACK Dhcp4Callback;
    VOID *CallbackContext;
    UINT32 OptionCount;
    EFI_DHCP4_PACKET_OPTION **OptionList;
} EFI_DHCP4_CONFIG_DATA;

DiscoverTryCount

Number of times to try sending a packet during the Dhcp4SendDiscover event and waiting for a response during the Dhcp4RcvdOffer event. (This value is also the number of entries in
the DiscoverTimeout array.) Set to zero to use the default try counts and timeout values.

DiscoverTimeout  Maximum amount of time (in seconds) to wait for returned packets in each of the retries. Timeout values of zero will default to a timeout value of one second. Set to NULL to use default timeout values.

RequestTryCount  Number of times to try sending a packet during the Dhcp4SendRequest event and waiting for a response during the Dhcp4RcvdAck event before accepting failure. (This value is also the number of entries in the RequestTimeout array.) Set to zero to use the default try counts and timeout values.

RequestTimeout  Maximum amount of time (in seconds) to wait for return packets in each of the retries. Timeout values of zero will default to a timeout value of one second. Set to NULL to use default timeout values.

ClientAddress  For a DHCPDISCOVER, setting this parameter to the previously allocated IP address will cause the EFI DHCPv4 Protocol driver to enter the Dhcp4InitReboot state. Also, set this field to 0.0.0.0 to enter the Dhcp4Init state. For a DHCPINFORM this parameter should be set to the client network address which was assigned to the client during a DHCPDISCOVER.

Dhcp4Callback  The callback function to intercept various events that occurred in the DHCP configuration process. Set to NULL to ignore all those events. Type EFI_DHCP4_CALLBACK is defined below.

CallbackContext  Pointer to the context that will be passed to Dhcp4Callback when it is called.

OptionCount  Number of DHCP options in the OptionList.

OptionList  List of DHCP options to be included in every packet that is sent during the Dhcp4SendDiscover event. Pad options are appended automatically by DHCP driver in outgoing DHCP packets. If OptionList itself contains pad option, they are ignored by the driver. OptionList can be freed after EFI_DHCP4_PROTOCOL.Configure() returns. Ignored if OptionCount is zero. Type EFI_DHCP4_PACKET_OPTION is defined below.

//**********************************************************
// EFI_DHCP4_CALLBACK
//**********************************************************
typedef EFI_STATUS (*EFI_DHCP4_CALLBACK)(
    IN EFI_DHCP4_PROTOCOL *This,
    IN VOID *Context,
    IN EFI_DHCP4_STATE CurrentState,
    IN EFI_DHCP4_EVENT Dhcp4Event,
    IN EFI_DHCP4_PACKET *Packet, OPTIONAL
    OUT EFI_DHCP4_PACKET **NewPacket OPTIONAL
);
This

Pointer to the EFI DHCPv4 Protocol instance that is used to configure this callback function.

Context

Pointer to the context that is initialized by `EFI_DHCP4_PROTOCOL.Configure()`.

currentState

The current operational state of the EFI DHCPv4 Protocol driver. Type `EFI_DHCP4_STATE` is defined in `EFI_DHCP4_PROTOCOL.GetModeData()`.

Dhcp4Event

The event that occurs in the current state, which usually means a state transition. Type `EFI_DHCP4_EVENT` is defined below.

Packet

The DHCP packet that is going to be sent or already received. May be `NULL` if the event has no associated packet. Do not cache this packet except for copying it. Type `EFI_DHCP4_PACKET` is defined in `EFI_DHCP4_PROTOCOL.GetModeData()`.

NewPacket

The packet that is used to replace the above `Packet`. Do not set this pointer exactly to the above `Packet` or a modified `Packet`. `NewPacket` can be `NULL` if the EFI DHCPv4 Protocol driver does not expect a new packet to be returned. The user may set `*NewPacket` to `NULL` if no replacement occurs.

`EFI_DHCP4_CALLBACK` is provided by the consumer of the EFI DHCPv4 Protocol driver to intercept events that occurred in the configuration process. This structure provides advanced control of each state transition of the DHCP process. The returned status code determines the behavior of the EFI DHCPv4 Protocol driver. There are three possible returned values, which are described in the following table.

<table>
<thead>
<tr>
<th>Status Code</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>EFI_SUCCESS</code></td>
<td>Tells the EFI DHCPv4 Protocol driver to continue the DHCP process. When</td>
</tr>
<tr>
<td></td>
<td>it is in the <code>Dhcp4Selecting</code> state, it tells the EFI DHCPv4 Protocol</td>
</tr>
<tr>
<td></td>
<td>driver to stop collecting additional packets. The driver will exit the</td>
</tr>
<tr>
<td></td>
<td><code>Dhcp4Selecting</code> state and enter the <code>Dhcp4Requesting</code> state.</td>
</tr>
<tr>
<td><code>EFI_NOT_READY</code></td>
<td>Only used in the <code>Dhcp4Selecting</code> state. The EFI DHCPv4 Protocol driver</td>
</tr>
<tr>
<td></td>
<td>will continue to wait for more packets until the retry timeout</td>
</tr>
<tr>
<td></td>
<td>expires.</td>
</tr>
<tr>
<td><code>EFI_ABORTED</code></td>
<td>Tells the EFI DHCPv4 Protocol driver to abort the current process and</td>
</tr>
<tr>
<td></td>
<td>return to the <code>Dhcp4Init</code> or <code>Dhcp4InitReboot</code> state.</td>
</tr>
</tbody>
</table>
typedef enum {
    Dhcp4SendDiscover    = 0x01,
    Dhcp4RcvdOffer       = 0x02,
    Dhcp4SelectOffer     = 0x03,
    Dhcp4SendRequest     = 0x04,
    Dhcp4RcvdAck         = 0x05,
    Dhcp4RcvdNak         = 0x06,
    Dhcp4SendDecline     = 0x07,
    Dhcp4BoundCompleted  = 0x08,
    Dhcp4EnterRenewing   = 0x09,
    Dhcp4EnterRebinding  = 0x0a,
    Dhcp4AddressLost     = 0x0b,
    Dhcp4Fail            = 0x0c
} EFI_DHCP4_EVENT;

Following is a description of the fields in the above enumeration.

**Dhcp4SendDiscover** The packet to start the configuration sequence is about to be sent. The packet is passed to Dhcp4Callback and can be modified or replaced in Dhcp4Callback.

**Dhcp4RcvdOffer** A reply packet was just received. This packet is passed to Dhcp4Callback, which may copy this packet and cache it for selecting a task later. If the callback returns EFI_SUCCESS, this driver will finish the selecting state. If EFI_NOT_READY is returned, this driver will continue to wait for additional reply packets until the timer expires. In either case, Dhcp4SelectOffer will occur for the user to select an offer.

**Dhcp4SelectOffer** It is time for Dhcp4Callback to select an offer. This driver passes the latest received DHCPOFFER packet to the callback. The Dhcp4Callback may store one packet in the NewPacket parameter of the function that was selected from previously received DHCPOFFER packets. If the latest packet is the selected one or if the user does not care about it, no extra overhead is needed. Simply skipping this event is enough.

**Dhcp4SendRequest** A request packet is about to be sent. The user can modify or replace this packet.

**Dhcp4RcvdAck** A DHCPACK packet was received and will be passed to Dhcp4Callback. The callback may decline this DHCPACK packet by returning EFI_ABORTED. In this case, the EFI DHCPv4 Protocol driver will proceed to the Dhcp4SendDecline event.

**Dhcp4RcvdNak** A DHCPNAK packet was received and will be passed to Dhcp4Callback. The EFI DHCPv4 Protocol driver will then return to the Dhcp4Init state no matter what status code is returned from the callback function.
Dhcp4SendDecline A decline packet is about to be sent. Dhcp4Callback can modify or replace this packet. The EFI DHCPv4 Protocol driver will then be set to the Dhcp4Init state.

Dhcp4BoundCompleted The DHCP configuration process has completed. No packet is associated with this event.

Dhcp4EnterRenewing It is time to enter the Dhcp4Renewing state and to contact the server that originally issued the network address. No packet is associated with this event.

Dhcp4EnterRebinding It is time to enter the Dhcp4Rebinding state and to contact any server. No packet is associated with this event.

Dhcp4AddressLost The configured IP address was lost either because the lease has expired, the user released the configuration, or a DHCPNAK packet was received in the Dhcp4Renewing or Dhcp4Rebinding state. No packet is associated with this event.

Dhcp4Fail The DHCP process failed because a DHCPNAK packet was received or the user aborted the DHCP process at a time when the configuration was not available yet. No packet is associated with this event.

//*******************************************
// EFI_DHCP4_HEADER
//*******************************************
#pragma pack(1)
typedef struct{
    UINT8(Op Code);
    UINT8(Hw Type);
    UINT8(Hw AddrLen);
    UINT8(Hops);
    UINT32(Xid);
    UINT16(Seconds);
    UINT16(Reserved);
    EFI_IPv4_ADDRESS(ClientAddr);
    EFI_IPv4_ADDRESS(YourAddr);
    EFI_IPv4_ADDRESS(ServerAddr);
    EFI_IPv4_ADDRESS(GatewayAddr);
    UINT8(ClientHwAddr[16]);
    CHAR8(ServerName[64]);
    CHAR8(BootFileName[128]);
} EFI_DHCP4_HEADER;
#pragma pack()
Seconds  Number of seconds that have elapsed since the client began address
acquisition or the renewal process.
Reserved   Reserved for future use.
ClientAddr Client IP address from the client.
YourAddr   Client IP address from the server.
ServerAddr IP address of the next server in bootstrap.
GatewayAddr Relay agent IP address.
ClientHwAddr Client hardware address.
ServerName Optional server host name.
BootFileName Boot file name.

EFI_DHCP4_HEADER describes the semantics of the DHCP packet header. This packet header is in
network byte order.

```c
//*******************************************
// EFI_DHCP4_PACKET_OPTION
//*******************************************
#pragma pack(1)
typedef struct {
    UINT8    OpCode;
    UINT8    Length;
    UINT8[1] Data;
} EFI_DHCP4_PACKET_OPTION;
#pragma pack()
```

OpCode   DHCP option code.
Length   Length of the DHCP option data. Not present if OpCode is 0 or 255.
Data     Start of the DHCP option data. Not present if OpCode is 0 or 255 or if
Length is zero.

The DHCP packet option data structure is used to reference option data that is packed in the DHCP
packets. Use caution when accessing multibyte fields because the information in the DHCP packet may
not be properly aligned for the machine architecture.

**Status Codes Returned**

<table>
<thead>
<tr>
<th>Code</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>EFI_SUCCESS</td>
<td>The EFI DHCPv4 Protocol driver is now in the Dhcp4Init or Dhcp4InitReboot state, if the original state of this driver was Dhcp4Stopped, Dhcp4Init, Dhcp4InitReboot, or Dhcp4Bound and the value of Dhcp4CfgData was not NULL. Otherwise, the state was left unchanged.</td>
</tr>
<tr>
<td>EFI_ACCESS_DENIED</td>
<td>This instance of the EFI DHCPv4 Protocol driver was not in the Dhcp4Stopped, Dhcp4Init, Dhcp4InitReboot, or Dhcp4Bound state.</td>
</tr>
<tr>
<td>EFI_ACCESS_DENIED</td>
<td>Another instance of this EFI DHCPv4 Protocol driver is already in a valid configured state.</td>
</tr>
</tbody>
</table>

UEFI Forum, Inc.  March 2021  1464
 EFI_DHCP4_PROTOCOL.Start()

Summary
Starts the DHCP configuration process.

Prototype
typedef
EFI_STATUS
(EIFIAPI *EFI_DHCP4_START) (  
IN EFI_DHCP4_PROTOCOL *This,
IN EFI_EVENT CompletionEvent OPTIONAL );

Parameters
This Pointer to the EFI_DHCP4_PROTOCOL instance.
CompletionEvent If not NULL, indicates the event that will be signaled when the EFI DHCPv4 Protocol driver is transferred into the Dhc4Bound state or when the DHCP process is aborted.
EFI_DHCP4_PROTOCOL.GetModeData() can be called to check the completion status. If NULL, EFI_DHCP4_PROTOCOL.Start() will wait until the driver is transferred into the Dhc4Bound state or the process fails.

Description
The Start() function starts the DHCP configuration process. This function can be called only when the EFI DHCPv4 Protocol driver is in the Dhcp4Init or Dhcp4InitReboot state.

If the DHCP process completes successfully, the state of the EFI DHCPv4 Protocol driver will be transferred through Dhc4Selecting and Dhc4Requesting to the Dhc4Bound state. The CompletionEvent will then be signaled if it is not NULL.

If the process aborts, either by the user or by some unexpected network error, the state is restored to the Dhcp4Init state. The Start() function can be called again to restart the process.

Refer to RFC 2131 for precise state transitions during this process. At the time when each event occurs in this process, the callback function that was set by EFI_DHCP4_PROTOCOL.Configure() will be called and the user can take this opportunity to control the process.
Status Codes Returned

<table>
<thead>
<tr>
<th>Code</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>EFI_SUCCESS</td>
<td>The DHCP configuration process has started, or it has completed when CompletionEvent is NULL.</td>
</tr>
<tr>
<td>EFI_NOT_STARTED</td>
<td>The EFI DHCPv4 Protocol driver is in the Dhcp4Stopped state. EFI_DHCP4_PROTOCOL.Configure() needs to be called.</td>
</tr>
<tr>
<td>EFI_INVALID_PARAMETER</td>
<td>This is NULL.</td>
</tr>
<tr>
<td>EFI_OUT_OF_RESOURCES</td>
<td>Required system resources could not be allocated.</td>
</tr>
<tr>
<td>EFI_TIMEOUT</td>
<td>The DHCP configuration process failed because no response was received from the server within the specified timeout value.</td>
</tr>
<tr>
<td>EFI_ABORTED</td>
<td>The user aborted the DHCP process.</td>
</tr>
<tr>
<td>EFI_ALREADY_STARTED</td>
<td>Some other EFI DHCPv4 Protocol instance already started the DHCP process.</td>
</tr>
<tr>
<td>EFI_DEVICE_ERROR</td>
<td>An unexpected network or system error occurred.</td>
</tr>
<tr>
<td>EFI_NO_MEDIA</td>
<td>There was a media error.</td>
</tr>
</tbody>
</table>

**EFI_DHCP4_PROTOCOL.RenewRebind()**

Summary

Extends the lease time by sending a request packet.

Prototype

```c
typedef EFI_STATUS (EFIAPIC *EFI_DHCP4_RENEW_REBIND) (  
   IN EFI_DHCP4_PROTOCOL *This,
   IN BOOLEAN        RebindRequest,
   IN EFI_EVENT      CompletionEvent OPTIONAL
);
```

Parameters

- **This**: Pointer to the EFI_DHCP4_PROTOCOL instance.
- **RebindRequest**: If TRUE, this function broadcasts the request packets and enters the Dhcp4Rebinding state. Otherwise, it sends a unicast request packet and enters the Dhcp4Renewing state.
- **CompletionEvent**: If not NULL, this event is signaled when the renew/rebind phase completes or some error occurs. EFI_DHCP4_PROTOCOL.GetModeData() can be called to check the completion status. If NULL, EFI_DHCP4_PROTOCOL.RenewRebind() will busy-wait until the DHCP process finishes.

Description

The **RenewRebind()** function is used to manually extend the lease time when the EFI DHCPv4 Protocol driver is in the Dhcp4Bound state and the lease time has not expired yet. This function will send a request packet to the previously found server (or to any server when RebindRequest is TRUE) and transfer the
state into the **Dhcp4Renewing** state (or **Dhcp4Rebinding** when **RebindingRequest** is **TRUE**). When a response is received, the state is returned to **Dhcp4Bound**.

If no response is received before the try count is exceeded (the **RequestTryCount** field that is specified in **EFI_DHCP4_CONFIG_DATA**) but before the lease time that was issued by the previous server expires, the driver will return to the **Dhcp4Bound** state and the previous configuration is restored. The outgoing and incoming packets can be captured by the **EFI_DHCP4_CALLBACK** function.

**Status Codes Returned**

<table>
<thead>
<tr>
<th>Code</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>EFI_SUCCESS</strong></td>
<td>The EFI DHCPv4 Protocol driver is now in the <strong>Dhcp4Renewing</strong> state or is back to the <strong>Dhcp4Bound</strong> state.</td>
</tr>
<tr>
<td><strong>EFI_NOT_STARTED</strong></td>
<td>The EFI DHCPv4 Protocol driver is in the <strong>Dhcp4Stopped</strong> state. <strong>EFI_DHCP4_PROTOCOL.Configure()</strong> needs to be called.</td>
</tr>
<tr>
<td><strong>EFI_INVALID_PARAMETER</strong></td>
<td><strong>This</strong> is <strong>NULL</strong>.</td>
</tr>
<tr>
<td><strong>EFI_TIMEOUT</strong></td>
<td>There was no response from the server when the try count was exceeded.</td>
</tr>
<tr>
<td><strong>EFI_ACCESS_DENIED</strong></td>
<td>The driver is not in the <strong>Dhcp4Bound</strong> state.</td>
</tr>
<tr>
<td><strong>EFIDEVICE_ERROR</strong></td>
<td>An unexpected network or system error occurred.</td>
</tr>
</tbody>
</table>

**EFI_DHCP4_PROTOCOL.Release()**

**Summary**
Releases the current address configuration.

**Prototype**
```c
typedef
EFI_STATUS
(EIFI_API *EFI_DHCP4_RELEASE) (|
IN EFI_DHCP4_PROTOCOL *This |
);
```

**Parameters**

| This | Pointer to the **EFI_DHCP4_PROTOCOL** instance. |

**Description**
The **Release()** function releases the current configured IP address by doing either of the following:

- Sending a DHCPRELEASE packet when the EFI DHCPv4 Protocol driver is in the **Dhcp4Bound** state
- Setting the previously assigned IP address that was provided with the **EFI_DHCP4_PROTOCOL.Configure()** function to 0.0.0.0 when the driver is in the **Dhcp4InitReboot** state

After a successful call to this function, the EFI DHCPv4 Protocol driver returns to the **Dhcp4Init** state and any subsequent incoming packets will be discarded silently.
**Status Codes Returned**

<table>
<thead>
<tr>
<th>Code</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>EFI_SUCCESS</td>
<td>The EFI DHCPv4 Protocol driver is now in the <strong>Dhcp4Init</strong> phase.</td>
</tr>
<tr>
<td>EFI_INVALID_PARAMETER</td>
<td>This is <strong>NULL</strong>.</td>
</tr>
<tr>
<td>EFI_ACCESS_DENIED</td>
<td>The EFI DHCPv4 Protocol driver is not in the <strong>Dhcp4Bound</strong> or <strong>Dhcp4InitReboot</strong> state.</td>
</tr>
<tr>
<td>EFI_DEVICE_ERROR</td>
<td>An unexpected network or system error occurred.</td>
</tr>
</tbody>
</table>

**EFI_DHCP4_PROTOCOL.Stop()**

**Summary**

Stops the DHCP configuration process.

**Prototype**

```c
typedef
    EFI_STATUS
(EFI_API *EFI_DHCP4_STOP)(
    IN EFI_DHCP4_PROTOCOL    *This
);```

**Parameters**

- `This` Pointer to the **EFI_DHCP4_PROTOCOL** instance.

**Description**

The **Stop()** function is used to stop the DHCP configuration process. After this function is called successfully, the EFI DHCPv4 Protocol driver is transferred into the **Dhcp4Stopped** state. **EFI_DHCP4_PROTOCOL.Configure()** needs to be called before DHCP configuration process can be started again. This function can be called when the EFI DHCPv4 Protocol driver is in any state.

**Status Codes Returned**

<table>
<thead>
<tr>
<th>Code</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>EFI_SUCCESS</td>
<td>The EFI DHCPv4 Protocol driver is now in the <strong>Dhcp4Stopped</strong> state.</td>
</tr>
<tr>
<td>EFI_INVALID_PARAMETER</td>
<td>This is <strong>NULL</strong>.</td>
</tr>
<tr>
<td>EFI_NO_MEDIA</td>
<td>There was a media error.</td>
</tr>
</tbody>
</table>

**EFI_DHCP4_PROTOCOL.Build()**

**Summary**

Builds a DHCP packet, given the options to be appended or deleted or replaced.
Prototype

typedef
    EFI_STATUS
    (EFIAPI *EFI_DHCP4_BUILD) (  
        IN EFI_DHCP4_PROTOCOL    *This,
        IN EFI_DHCP4_PACKET      *SeedPacket,
        IN UINT32               DeleteCount,
        IN UINT8                *DeleteList  OPTIONAL,
        IN UINT32               AppendCount,
        IN EFI_DHCP4_PACKET_OPTION   *AppendList[] OPTIONAL,
        OUT EFI_DHCP4_PACKET     **NewPacket
    );

Parameters

This
    Pointer to the EFI_DHCP4_PROTOCOL instance.

SeedPacket
    Initial packet to be used as a base for building new packet. Type
    EFI_DHCP4_PACKET is defined in
    EFI_DHCP4_PROTOCOL.GetModeData().

DeleteCount
    Number of opcodes in the DeleteList.

DeleteList
    List of opcodes to be deleted from the seed packet. Ignored if
    DeleteCount is zero.

AppendCount
    Number of entries in the OptionList.

AppendList
    Pointer to a DHCP option list to be appended to SeedPacket. If
    SeedPacket also contains options in this list, they are replaced by
    new options (except pad option). Ignored if AppendCount is zero.
    Type EFI_DHCP4_PACKET_OPTION is defined in
    EFI_DHCP4_PROTOCOL.Configure().

NewPacket
    Pointer to storage for the pointer to the new allocated packet. Use
    the EFI Boot Service FreePool() on the resulting pointer when
    done with the packet.

Description

The Build() function is used to assemble a new packet from the original packet by replacing or deleting
existing options or appending new options. This function does not change any state of the EFI DHCPv4
Protocol driver and can be used at any time.
Status Codes Returned

<table>
<thead>
<tr>
<th>Code</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>EFI_SUCCESS</td>
<td>The new packet was built.</td>
</tr>
<tr>
<td>EFI_OUT_OF_RESOURCES</td>
<td>Storage for the new packet could not be allocated.</td>
</tr>
<tr>
<td>EFI_INVALID_PARAMETER</td>
<td>One or more of the following conditions is TRUE:</td>
</tr>
<tr>
<td></td>
<td>• This is NULL</td>
</tr>
<tr>
<td></td>
<td>• SeedPacket is NULL</td>
</tr>
<tr>
<td></td>
<td>• SeedPacket is not a well-formed DHCP packet.</td>
</tr>
<tr>
<td></td>
<td>• AppendCount is not zero and AppendList is NULL.</td>
</tr>
<tr>
<td></td>
<td>• DeleteCount is not zero and DeleteList is NULL.</td>
</tr>
<tr>
<td></td>
<td>• NewPacket is NULL</td>
</tr>
<tr>
<td></td>
<td>• Both DeleteCount and AppendCount are zero and NewPacket is not NULL.</td>
</tr>
</tbody>
</table>

EFI_DHCP4_PROTOCOL.TransmitReceive()

Summary
Transmits a DHCP formatted packet and optionally waits for responses.

Prototype
```c
typedef EFI_STATUS
    (EFIAPI *EFI_DHCP4_TRANSMIT_RECEIVE) (    
    IN EFI_DHCP4_PROTOCOL     *This,    
    IN EFI_DHCP4_TRANSMIT_RECEIVE_TOKEN *Token
    );
```

Parameters
- **This** Pointer to the EFI_DHCP4_PROTOCOL instance.
- **Token** Pointer to the EFI_DHCP4_TRANSMIT_RECEIVE_TOKEN structure.

Type **EFI_DHCP4_TRANSMIT_RECEIVE_TOKEN** is defined in “Related Definitions” below.

Description
The TransmitReceive() function is used to transmit a DHCP packet and optionally wait for the response from servers. This function does not change the state of the EFI DHCPv4 Protocol driver and thus can be used at any time.
Related Definitions

```c
typedef struct {
    EFI_STATUS     Status;
    EFI_EVENT     CompletionEvent;
    EFI_IPv4_ADDRESS RemoteAddress;
    UINT16        RemotePort;
    EFI_IPv4_ADDRESS GatewayAddress;
    UINT32        ListenPointCount;
    EFI_DHCP4_LISTEN_POINT *ListenPoints;
    UINT32        TimeoutValue;
    EFI_DHCP4_PACKET   *Packet;
    UINT32        ResponseCount;
    EFI_DHCP4_PACKET   *ResponseList;
} EFI_DHCP4_TRANSMIT_RECEIVE_TOKEN;
```

**Status**  
The completion status of transmitting and receiving. Possible values are described in the “Status Codes Returned” table below. When `CompletionEvent` is `NULL`, this status is the same as the one returned by the `TransmitReceive()` function.

**CompletionEvent**  
If not `NULL`, the event that will be signaled when the collection process completes. If `NULL`, this function will busy-wait until the collection process completes.

**RemoteAddress**  
Pointer to the server IP address. This address may be a unicast, multicast, or broadcast address.

**RemotePort**  
Server listening port number. If zero, the default server listening port number (67) will be used.

**GatewayAddress**  
Pointer to the gateway address to override the existing setting.

**ListenPointCount**  
The number of entries in `ListenPoints`. If zero, the default station address and port number 68 are used.

**ListenPoints**  
An array of station address and port number pairs that are used as receiving filters. The first entry is also used as the source address and source port of the outgoing packet. Type `EFI_DHCP4_LISTEN_POINT` is defined below.

**TimeoutValue**  
Number of seconds to collect responses. Zero is invalid.

**Packet**  
Pointer to the packet to be transmitted. Type `EFI_DHCP4_PACKET` is defined in `EFI_DHCP4_PROTOCOL.GetModeData()`.

**ResponseCount**  
Number of received packets.

**ResponseList**  
Pointer to the allocated list of received packets. The caller must use the EFI Boot Service `FreePool()` when done using the received packets.
typedef struct {
  EFI_IPv4_ADDRESS  ListenAddress;
  EFI_IPv4_ADDRESS  SubnetMask;
  UINT16            ListenPort;
} EFI_DHCP4_LISTEN_POINT;

**ListenAddress**
Alternate listening address. It can be a unicast, multicast, or broadcast address. The `TransmitReceive()` function will collect only those packets that are destined to this address.

**SubnetMask**
The subnet mask of above listening unicast/broadcast IP address. Ignored if `ListenAddress` is a multicast address. If it is `0.0.0.0`, the subnet mask is automatically computed from unicast `ListenAddress`. Cannot be `0.0.0.0` if `ListenAddress` is direct broadcast address on subnet.

**ListenPort**
Alternate station source (or listening) port number. If zero, then the default station port number (68) will be used.

### Status Codes Returned

<table>
<thead>
<tr>
<th>Status Code</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>EFI_SUCCESS</td>
<td>The packet was successfully queued for transmission.</td>
</tr>
<tr>
<td>EFI_INVALID_PARAMETER</td>
<td>One or more of the following conditions is <strong>TRUE:</strong></td>
</tr>
<tr>
<td></td>
<td>• <strong>This</strong> is <strong>NULL</strong>.</td>
</tr>
<tr>
<td></td>
<td>• <strong>Token.RemoteAddress</strong> is zero.</td>
</tr>
<tr>
<td></td>
<td>• <strong>Token.Packet</strong> is <strong>NULL</strong>.</td>
</tr>
<tr>
<td></td>
<td>• <strong>Token.Packet</strong> is not a well-formed DHCP packet.</td>
</tr>
<tr>
<td></td>
<td>• The transaction ID in <strong>Token.Packet</strong> is in use by another DHCP process.</td>
</tr>
<tr>
<td>EFI_NOT_READY</td>
<td>The previous call to this function has not finished yet. Try to call this function after collection process completes.</td>
</tr>
<tr>
<td>EFI_NO_MAPPING</td>
<td>The default station address is not available yet.</td>
</tr>
<tr>
<td>EFI_OUT_OF_RESOURCES</td>
<td>Required system resources could not be allocated.</td>
</tr>
<tr>
<td>EFI_UNSUPPORTED</td>
<td>The implementation doesn't support this function</td>
</tr>
<tr>
<td>EFI_NO_MEDIA</td>
<td>There was a media error.</td>
</tr>
<tr>
<td>Others</td>
<td>Some other unexpected error occurred.</td>
</tr>
</tbody>
</table>

### EFI_DHCP4_PROTOCOL.Parse()

**Summary**

Parses the packed DHCP option data.
Prototype

typedef
 EFI_STATUS
 (EFI_API EFI_DHCP4_PARSE) (  
 IN EFI_DHCP4_PROTOCOL *This,
 IN EFI_DHCP4_PACKET *Packet
 IN OUT UINT32 *OptionCount,
 IN OUT EFI_DHCP4_PACKET_OPTION *PacketOptionList[] OPTIONAL
);

Parameters

This Pointer to the EFI_DHCP4_PROTOCOL instance.
Packet Pointer to packet to be parsed. Type EFI_DHCP4_PACKET is defined in EFI_DHCP4_PROTOCOL.GetModeData().
OptionCount On input, the number of entries in the PacketOptionList. On output, the number of entries that were written into the PacketOptionList.
PacketOptionList List of packet option entries to be filled in. End option or pad options are not included. Type EFI_DHCP4_PACKET_OPTION is defined in EFI_DHCP4_PROTOCOL.Configure().

Description

The Parse() function is used to retrieve the option list from a DHCP packet. If *OptionCount isn’t zero, and there is enough space for all the DHCP options in the Packet, each element of PacketOptionList is set to point to somewhere in the Packet->Dhcp4.Option where a new DHCP option begins. If RFC3396 is supported, the caller should reassemble the parsed DHCP options to get the final result. If *OptionCount is zero or there isn’t enough space for all of them, the number of DHCP options in the Packet is returned in OptionCount.

Status Codes Returned

<table>
<thead>
<tr>
<th>Code</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>EFI_SUCCESS</td>
<td>The packet was successfully parsed.</td>
</tr>
<tr>
<td>EFI_INVALID_PARAMETER</td>
<td>One or more of the following conditions is TRUE:</td>
</tr>
<tr>
<td></td>
<td>• This is NULL.</td>
</tr>
<tr>
<td></td>
<td>• Packet is NULL.</td>
</tr>
<tr>
<td></td>
<td>• Packet is not a well-formed DHCP packet.</td>
</tr>
<tr>
<td></td>
<td>• OptionCount is NULL.</td>
</tr>
<tr>
<td>EFI_BUFFER_TOO_SMALL</td>
<td>One or more of the following conditions is TRUE:</td>
</tr>
<tr>
<td></td>
<td>• *OptionCount is smaller than the number of options that were found in the Packet.</td>
</tr>
<tr>
<td></td>
<td>• PacketOptionList is NULL.</td>
</tr>
<tr>
<td>EFI_OUT_OF_RESOURCE</td>
<td>The packet is failed to parse because of resource shortage.</td>
</tr>
</tbody>
</table>
29.3 EFI DHCP6 Protocol

This section provides a detailed description of the EFI_DHCP6_PROTOCOL and the EFI_DHCP6_SERVICE_BINDING_PROTOCOL.

29.3.1 DHCP6 Service Binding Protocol

EFI_DHCP6_SERVICE_BINDING_PROTOCOL

Summary

The EFI DHCPv6 Service Binding Protocol is used to locate communication devices that are supported by an EFI DHCPv6 Protocol driver and to create and destroy EFI DHCPv6 Protocol child instances that can use the underlying communications device.

GUID

#define EFI_DHCP6_SERVICE_BINDING_PROTOCOL _GUID \\
{0x9fb9a8a1,0x2f4a,0x43a6,\}
{0x88,0x9c,0xd0,0xf7,0xb6,0xc4,0x7a,0xd5}

Description

A network application or driver that requires basic DHCPv6 services can use one of the protocol handler services, such as BS->LocateHandleBuffer(), to search for devices that publish an EFI DHCPv6 Service Binding Protocol GUID. Each device with a published EFI DHCPv6 Service Binding Protocol GUID supports the EFI DHCPv6 Protocol and may be available for use.

After a successful call to the EFI_DHCP6_SERVICE_BINDING_PROTOCOL.CreateChild() function, the newly created EFI DHCPv6 Protocol child instance is ready to be used by a network application or driver.

Before a network application or driver terminates execution, every successful call to the EFI_DHCP6_SERVICE_BINDING_PROTOCOL.CreateChild() function must be matched with a call to the EFI_DHCP6_SERVICE_BINDING_PROTOCOL.DestroyChild() function.

29.3.2 DHCP6 Protocol

EFI_DHCP6_PROTOCOL

Summary

The EFI DHCPv6 Protocol is used to get IPv6 addresses and other configuration parameters from DHCPv6 servers.
GUID

```c
#define EFI_DHCP6_PROTOCOL_GUID  \{
    0x87c8bad7, 0x595, 0x4053, 0x82, 0x97, 0xde, 0xde, 0x39, 0x5f, 0x5d, 0x5b
```

Protocol Interface Structure

```c
typedef struct _EFI_DHCP6_PROTOCOL  {
    EFI_DHCP6_GET_MODE_DATA GetModeData;
    EFI_DHCP6_CONFIGURE Configure;
    EFI_DHCP6_START Start;
    EFI_DHCP6_INFO_REQUEST InfoRequest;
    EFI_DHCP6_RENEW_REBIND RenewRebind;
    EFI_DHCP6_DECLINE Decline;
    EFI_DHCP6_RELEASE Release;
    EFI_DHCP6_STOP Stop;
    EFI_DHCP6_PARSE Parse;
} EFI_DHCP6_PROTOCOL;
```

Parameters

**GetModeData**

Get the current operating mode data and configuration data for the EFI DHCPv6 Protocol instance. See the `GetModeData()` function description.

**Configure**

Initialize or clean up the configuration data for the EFI DHCPv6 Protocol instance. See the `Configure()` function description.

**Start**

Start the DHCPv6 S.A.R.R process. See the `Start()` function description.

**InfoRequest**

Request configuration parameters without the assignment of any IPv6 addresses to the client. See the `InfoRequest()` function description.

**RenewRebind**

Tries to manually extend the valid and preferred lifetimes for the IPv6 addresses of the configured IA by sending Renew or Rebind packet. See the `RenewRebind()` function description.

**Decline**

Inform that one or more addresses assigned by a DHCPv6 server are already in use by another node. See the `Decline()` function description.

**Release**

Release one or more addresses associated with the configured IA. See the `Release()` function description.

**Stop**

Stop the DHCPv6 S.A.R.R process. See the `Stop()` function description.
**Parse**

Parses the option data in the DHCPv6 packet. See the `Parse()` function description.

**Description**

The EFI DHCPv6 Protocol is used to get IPv6 addresses and other configuration parameters from DHCPv6 servers.

**Note:** Byte Order: All the IPv6 addresses that are described in `EFI_DHCPv6_PROTOCOL` are stored in network byte order. Both incoming and outgoing DHCPv6 packets are also in network byte order. All other parameters that are defined in functions or data structures are stored in host byte order.

**EFI_DHCPv6_PROTOCOL.GetModeData()**

**Summary**

Retrieve the current operating mode data and configuration data for the EFI DHCPv6 Protocol instance.

**Prototype**

```c
typedef EFI_STATUS
  (EFIAPI *EFI_DHCP6_GET_MODE_DATA)(
   IN EFI_DHCP6_PROTOCOL *This,
   OUT EFI_DHCP6_MODE_DATA *Dhcp6ModeData OPTIONAL,
   OUT EFI_DHCP6_CONFIG_DATA *Dhcp6ConfigData OPTIONAL);
```

**Parameters**

- **This**
  Pointer to the `EFI_DHCP6_PROTOCOL` instance.

- **Dhcp6ModeData**
  Pointer to the DHCPv6 mode data structure. The caller is responsible for freeing this structure and each reference buffer. Type `EFI_DHCP6_MODE_DATA` is defined in “Related Definitions” below.

- **Dhcp6ConfigData**
  Pointer to the DHCPv6 configuration data structure. The caller is responsible for freeing this structure and each reference buffer. Type `EFI_DHCP6_CONFIG_DATA` is defined in `EFI_DHCP6_PROTOCOL.Configure()`.

**Description**

Retrieve the current operating mode data and configuration data for the EFI DHCPv6 Protocol instance.
Related Definitions

```c
//***************************************************************
// EFI_DHCP6_MODE_DATA
//***************************************************************
typedef struct {
    EFI_DHCP6_DUID   *ClientId;
    EFI_DHCP6_IA     *Ia;
} EFI_DHCP6_MODE_DATA;

ClientId
Pointer to the DHCPv6 unique identifier. The caller is responsible for freeing this buffer. Type EFI_DHCP6_DUID is defined below.

Ia
Pointer to the configured IA of current instance. The caller can free this buffer after using it. Type EFI_DHCP6_IA is defined below.

//***************************************************************
// EFI_DHCP6_DUID
//***************************************************************
typedef struct {
    UINT16    Length;
    UINT8     Duid[1];
} EFI_DHCP6_DUID;

Length
Length of DUID in octets.

Duid
Array of DUID octets.

The EFI_DHCP6_DUID structure is to specify DHCPv6 unique identifier for either DHCPv6 client or DHCPv6 server. The DUID-UUID shall be used for all transactions.

//***************************************************************
// EFI_DHCP6_IA
//***************************************************************
typedef struct {
    EFI_DHCP6_IA_DESCRIPTOR Descriptor;
    EFI_DHCP6_STATE     State;
    EFI_DHCP6_PACKET    *ReplyPacket;
    UINT32    IaAddressCount;
    EFI_DHCP6_IA_ADDRESS IaAddress[1];
} EFI_DHCP6_IA;
```
Descriptor

The descriptor for IA. Type `EFI_DHCP6_IA_DESCRIPTOR` is defined below.

State

The state of the configured IA. Type `EFI_DHCP6_STATE` is defined below.

ReplyPacket

Pointer to the cached latest Reply packet. May be `NULL` if no packet is cached.

IaAddressCount

Number of IPv6 addresses of the configured IA.

IaAddress

List of the IPv6 addresses of the configured IA. When the state of the configured IA is in `Dhcp6Bound`, `Dhcp6Renewing` and `Dhcp6Rebinding`, the IPv6 addresses are usable. Type `EFI_DHCP6_IA_ADDRESS` is defined below.

```c
typedef struct {
    UINT16 Type;
    UINT32 IaId;
} EFI_DHCP6_IA_DESCRIPTOR;
```

Type

Type for an IA.

IaId

The identifier for an IA.

#define EFI_DHCP6_IA_TYPE_NA 3
#define EFI_DHCP6_IA_TYPE_TA 4

`EFI_DHCP6_IA_TYPE_NA`

An IA which carries assigned not temporary address.

`EFI_DHCP6_IA_TYPE_TA`

An IA which carries assigned temporary address.
typedef enum {
    Dhcp6Init      = 0x0,
    Dhcp6Selecting = 0x1,
    Dhcp6Requesting = 0x2,
    Dhcp6Declining = 0x3,
    Dhcp6Confirming = 0x4,
    Dhcp6Releasing = 0x5,
    Dhcp6Bound     = 0x6,
    Dhcp6Renewing  = 0x7,
    Dhcp6Rebinding = 0x8
} EFI_DHCP6_STATE;

Table 29-2 describes the fields in the above enumeration.

**Table 29-2 Field Descriptions**

<table>
<thead>
<tr>
<th>Field</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Dhcp6Init</td>
<td>The EFI DHCPv6 Protocol instance is configured, and <code>start()</code> needs to be called</td>
</tr>
<tr>
<td>Dhcp6Selecting</td>
<td>A Solicit packet is sent out to discover DHCPv6 server, and the EFI DHCPv6 Protocol instance is collecting Advertise packets.</td>
</tr>
<tr>
<td>Dhcp6Requesting</td>
<td>A Request is sent out to the DHCPv6 server, and the EFI DHCPv6 Protocol instance is waiting for Reply packet.</td>
</tr>
<tr>
<td>Dhcp6Declining</td>
<td>A Decline packet is sent out to indicate one or more addresses of the configured IA are in use by another node, and the EFI DHCPv6 Protocol instance is waiting for Reply packet.</td>
</tr>
<tr>
<td>Dhcp6Confirming</td>
<td>A Confirm packet is sent out to confirm the IPv6 addresses of the configured IA, and the EFI DHCPv6 Protocol instance is waiting for Reply packet.</td>
</tr>
<tr>
<td>Dhcp6Releasing</td>
<td>A Release packet is sent out to release one or more IPv6 addresses of the configured IA, and the EFI DHCPv6 Protocol instance is waiting for Reply packet.</td>
</tr>
<tr>
<td>Dhcp6Bound</td>
<td>The DHCPv6 S.A.R.R process is completed for the configured IA.</td>
</tr>
<tr>
<td>Dhcp6Renewing</td>
<td>A Renew packet is sent out to extend lifetime for the IPv6 addresses of the configured IA, and the EFI DHCPv6 Protocol instance is waiting for Reply packet.</td>
</tr>
<tr>
<td>Dhcp6Rebinding</td>
<td>A Rebind packet is sent out to extend lifetime for the IPv6 addresses of the configured IA, and the EFI DHCPv6 Protocol instance is waiting for Reply packet.</td>
</tr>
</tbody>
</table>
typedef struct {
    EFI_IPv6_ADDRESS  IpAddress;
    UINT32           PreferredLifetime;
    UINT32           ValidLifetime;
} EFI_DHCP6_IA_ADDRESS;

IpAddress
    The IPv6 address.

PreferredLifetime
    The preferred lifetime in unit of seconds for the IPv6 address.

ValidLifetime
    The valid lifetime in unit of seconds for the IPv6 address.

The EFI_DHCP6_IA_ADDRESS structure is specify IPv6 address associated with an IA.

#pragma pack(1)
typedef struct {
    UINT32   Size;
    UINT32   Length;
    struct{
        EFI_DHCP6_HEADER Header;
        UINT8     Option[1];
    } Dhcp6;
} EFI_DHCP6_PACKET;
#pragma pack()
```c
//***********************************************************
// EFI_DHCP6_HEADER
//***********************************************************
#pragma pack(1)
typedef struct{
    UINT32 TransactionId:24;
    UINT32 MessageType:8;
} EFI_DHCP6_HEADER;
#pragma pack()

TransactionId
The DHCPv6 transaction ID.

MessageType
The DHCPv6 message type.

EFI_DHCP6_HEADER defines the format of the DHCPv6 header. See RFC 3315 for more information.

Status Codes Returned

<table>
<thead>
<tr>
<th>Code</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>EFI_SUCCESS</td>
<td>The mode data was returned.</td>
</tr>
<tr>
<td>EFI_ACCESS_DENIED</td>
<td>The EFI DHCPv6 Protocol instance has not been configured when Dhcp6ConfigData is not NULL.</td>
</tr>
</tbody>
</table>
| EFI_INVALID_PARAMETER | One or more following conditions are TRUE:
  • This is NULL
  • Both Dhcp6ConfigData and Dhcp6ModeData are NULL |

EFI_DHCP6_PROTOCOL.Configure ()

Summary
Initialize or clean up the configuration data for the EFI DHCPv6 Protocol instance.

Prototype

typedef EFI_STATUS
(EFIAPI *EFI_DHCP6_CONFIGURE) (  
   IN EFI_DHCP6_PROTOCOL *This,  
   IN EFI_DHCP6_CONFIG_DATA *Dhcp6CfgData OPTIONAL  
);

Parameters
This
Pointer to the EFI_DHCP6_PROTOCOL instance.
Dhcp6CfgData

Pointer to the DHCPv6 configuration data structure. Type EFI_DHCP6_CONFIG_DATA is defined in “Related Definitions” below.

Description

The **Configure()** function is used to initialize or clean up the configuration data of the EFI DHCPv6 Protocol instance.

- When Dhcp6CfgData is not **NULL** and **Configure()** is called successfully, the configuration data will be initialized in the EFI DHCPv6 Protocol instance and the state of the configured IA will be transferred into Dhcp6Init.
- When Dhcp6CfgData is **NULL** and **Configure()** is called successfully, the configuration data will be cleaned up and no IA will be associated with the EFI DHCPv6 Protocol instance.

To update the configuration data for an EFI DCHPv6 Protocol instance, the original data must be cleaned up before setting the new configuration data.

Related Definitions

```c
//***************************************************************
// EFI_DHCP6_CONFIG_DATA
//***************************************************************
typedef struct {
  EFI_DHCP6_CALLBACK Dhcp6Callback;
  VOID *CallbackContext;
  UINT32 OptionCount;
  EFI_DHCP6_PACKET_OPTION **OptionList;
  EFI_DHCP6_IA_DESCRIPTOR IaDescriptor;
  EFI_EVENT IaInfoEvent;
  BOOLEAN ReconfigureAccept;
  BOOLEAN RapidCommit;
  EFI_DHCP6_RETRANSMISSION *SolicitRetransmission;
} EFI_DHCP6_CONFIG_DATA;
```

**Dhcp6Callback**

The callback function is to intercept various events that occur in the DHCPv6 S.A.R.R process. Set to **NULL** to ignore all those events. Type EFI_DHCP6_CALLBACK is defined below.

**CallbackContext**

Pointer to the context that will be passed to Dhcp6Callback.

**OptionCount**

Number of the DHCPv6 options in the OptionList.

**OptionList**

List of the DHCPv6 options to be included in Solicit and Request packet. The buffer can be freed after EFI_DHCP6_PROTOCOL.Configure() returns. Ignored if OptionCount is zero. OptionList should not contain Client Identifier option and any
IA option, which will be appended by EFI DHCPv6 Protocol instance automatically. Type `EFI_DHCP6_PACKET_OPTION` is defined below.

IaDescriptor

The descriptor for the IA of the EFI DHCPv6 Protocol instance. Type `EFI_DHCP6_IA_DESCRIPTOR` is defined below.

IaInfoEvent

If not `NULL`, the event will be signaled when any IPv6 address information of the configured IA is updated, including IPv6 address, preferred lifetime and valid lifetime, or the DHCPv6 S.A.R.R process fails. Otherwise, `Start()`, `renewrebind()`, `decline()`, `release()` and `stop()` will be blocking operations, and they will wait for the exchange process completion or failure.

ReconfigureAccept

If `TRUE`, the EFI DHCPv6 Protocol instance is willing to accept Reconfigure packet. Otherwise, it will ignore it. Reconfigure Accept option can not be specified through `OptionList` parameter.

RapidCommit

If `TRUE`, the EFI DHCPv6 Protocol instance will send Solicit packet with Rapid Commit option. Otherwise, Rapid Commit option will not be included in Solicit packet. Rapid Commit option can not be specified through `OptionList` parameter.

SolicitRetransmission

Parameter to control Solicit packet retransmission behavior. Type `EFI_DHCP6_RETRANSMISSION` is defined in “Related Definition” below. The buffer can be freed after `EFI_DHCP6_PROTOCOL.Configure()` returns.

//**************************************************************
// EFI_DHCP6_CALLBACK
//**************************************************************
typedef EFI_STATUS (EFIAPI *EFI_DHCP6_CALLBACK)(
   IN EFI_DHCP6_PROTOCOL   *This,
   IN VOID          *Context,
   IN EFI_DHCP6_STATE   CurrentState,
   IN EFI_DHCP6_EVENT    Dhcp6Event,
   IN EFI_DHCP6_PACKET   *Packet,
   OUT EFI_DHCP6_PACKET  **NewPacket OPTIONAL
);

This

Pointer to the `EFI_DHCP6_PROTOCOL` instance that is used to configure this callback function.

Context

Pointer to the context that is initialized by `EFI_DHCP6_PROTOCOL.Configure()`.
CurrentState

The current state of the configured IA. Type \texttt{EFI\_DHCP6\_STATE} is defined in \texttt{EFI\_DHCP6\_PROTOCOL\_GetModeData()}. 

Dhcp6Event

The event that occurs in the current state, which usually means a state transition. Type \texttt{EFI\_DHCP6\_EVENT} is defined below. 

Packet

Pointer to the DHCPv6 packet that is about to be sent or has been received. The EFI DHCPv6 Protocol instance is responsible for freeing the buffer. Type \texttt{EFI\_DHCP6\_PACKET} is defined in \texttt{EFI\_DHCP6\_PROTOCOL\_GetModeData()}. 

NewPacket

Pointer to the new DHCPv6 packet to overwrite the \texttt{Packet}. \texttt{NewPacket} can not share the buffer with \texttt{Packet}. If *\texttt{NewPacket} is not NULL, the EFI DHCPv6 Protocol instance is responsible for freeing the buffer. 

\texttt{EFI\_DHCP6\_CALLBACK} is provided by the consumer of the EFI DHCPv6 Protocol instance to intercept events that occurs in the DHCPv6 S.A.R.R process. There are two possible returned values, which are described in the following table. 

\begin{table}[h]
\centering
\begin{tabular}{|l|l|}
\hline
\texttt{EFI\_SUCCESS} & Tell the EFI DHCPv6 Protocol instance to continue the DHCPv6 S.A.R.R process. \\
\hline
\texttt{EFI\_ABORTED} & Tell the EFI DHCPv6 Protocol instance to abort the DHCPv6 S.A.R.R process, and the state of the configured IA will be transferred to \texttt{Dhcp6Init}. \\
\hline
\end{tabular}
\end{table}

//***************************************************
// \texttt{EFI\_DHCP6\_PACKET\_OPTION}
//***************************************************
#pragma pack(1)
typedef struct {
    UINT16     OpCode;
    UINT16     OpLen;
    UINT8      Data[1];
} \texttt{EFI\_DHCP6\_PACKET\_OPTION};
#pragma pack()

\texttt{OpCode}

The DHCPv6 option code, stored in network order.

\texttt{OpLen}

Length of the DHCPv6 option data, stored in network order. From the first byte to the last byte of the \texttt{Data} field.
Data

The data for the DHCPv6 option.

EFI_DHCP6_PACKET_OPTION defines the format of the DHCPv6 option, stored in network order. See RFC 3315 for more information. This data structure is used to reference option data that is packed in the DHCPv6 packet.

```
//***********************************************************************
// EFI_DHCP6_EVENT
//***********************************************************************
typedef enum {
    Dhcp6SendSolicit   = 0x0,
    Dhcp6RcvdAdvertise  = 0x1,
    Dhcp6SelectAdvertise = 0x2,
    Dhcp6SendRequest   = 0x3,
    Dhcp6RcvdReply    = 0x4,
    Dhcp6RcvdReconfigure = 0x5,
    Dhcp6SendDecline   = 0x6,
    Dhcp6SendConfirm   = 0x7,
    Dhcp6SendRelease   = 0x8,
    Dhcp6SendRenew     = 0x9,
    Dhcp6SendRebind    = 0xa
} EFI_DHCP6_EVENT;
```

Dhcp6SendSolicit

A Solicit packet is about to be sent. The packet is passed to Dhcp6Callback and can be modified or replaced in Dhcp6Callback.

Dhcp6RcvdAdvertise

An Advertise packet is received and will be passed to Dhcp6Callback.

Dhcp6SelectAdvertise

It is time for Dhcp6Callback to determine whether select the default Advertise packet by RFC 3315 policy, or overwrite it by specific user policy.

Dhcp6SendRequest

A Request packet is about to be sent. The packet is passed to Dhcp6Callback and can be modified or replaced in Dhcp6Callback.

Dhcp6RcvdReply

A Reply packet is received and will be passed to Dhcp6Callback.

Dhcp6RcvdReconfigure

A Reconfigure packet is received and will be passed to Dhcp6Callback.

Dhcp6SendDecline

A Decline packet is about to be sent. The packet is passed to Dhcp6Callback and can be modified or replaced in Dhcp6Callback.
Dhcp6SendConfirm

A Confirm packet is about to be sent. The packet is passed to Dhcp6Callback and can be modified or replaced in Dhcp6Callback.

Dhcp6SendRelease

A Release packet is about to be sent. The packet is passed to Dhcp6Callback and can be modified or replaced in Dhcp6Callback.

Dhcp6SendRenew

A Renew packet is about to be sent. The packet is passed to Dhcp6Callback and can be modified or replaced in Dhcp6Callback.

Dhcp6SendRebind

A Rebind packet is about to be sent. The packet is passed to Dhcp6Callback and can be modified or replaced in Dhcp6Callback.

//**********************************************************************
// EFI_DHCP6_RETRANSMISSION
//********************************************************************************
typedef struct {  
UINT32     Irt;  
UINT32     Mrc;  
UINT32     Mrt;  
UINT32     Mrd;  
} EFI_DHCP6_RETRANSMISSION;

Irt
Initial retransmission timeout.

Mrc
Maximum retransmission count for one packet. If Mrc is zero, there’s no upper limit for retransmission count.

Mrt
Maximum retransmission timeout for each retry. It’s the upper bound of the number of retransmission timeout. If Mrt is zero, there is no upper limit for retransmission timeout.

Mrd
Maximum retransmission duration for one packet. It’s the upper bound of the numbers the client may retransmit a message. If Mrd is zero, there’s no upper limit for retransmission duration.
Status Codes Returned

<table>
<thead>
<tr>
<th>Code</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>EFI_SUCCESS</td>
<td>The mode data was returned</td>
</tr>
<tr>
<td>EFI_INVALID_PARAMETER</td>
<td>One or more following conditions are TRUE</td>
</tr>
<tr>
<td></td>
<td>• This is <strong>NULL</strong></td>
</tr>
<tr>
<td></td>
<td>• OptionCount &gt; 0 and <strong>OptionList</strong> is <strong>NULL</strong></td>
</tr>
<tr>
<td></td>
<td>• <strong>OptionList</strong> is not <strong>NULL</strong>, and Client Id option, Reconfigure</td>
</tr>
<tr>
<td></td>
<td>Accept option, Rapid Commit option or any IA option is specified in the</td>
</tr>
<tr>
<td></td>
<td><strong>OptionList</strong></td>
</tr>
<tr>
<td></td>
<td>• <strong>IaDescriptor</strong>.Type is neither <strong>EFI_DHCP6_IA_TYPE_NA</strong> nor **</td>
</tr>
<tr>
<td></td>
<td>EFI_DHCP6_IA_TYPE_NA**</td>
</tr>
<tr>
<td></td>
<td>• <strong>IaDescriptor</strong> is not unique.</td>
</tr>
<tr>
<td></td>
<td>• Both <strong>IaInfoEvent</strong> and <strong>SolicitRetransmission</strong> are <strong>NULL</strong></td>
</tr>
<tr>
<td></td>
<td>• <strong>SolicitRetransmission</strong> is not <strong>NULL</strong>, and both <strong>SolicitRetransmission-&gt;Mrc</strong> and <strong>SolicitRetransmission-&gt;Mrd</strong> are zero.</td>
</tr>
<tr>
<td>EFI_ACCESS_DENIED</td>
<td>The EFI DHCPv6 Protocol instance has been already configured when <strong>Dhcp6CfgData</strong> is not <strong>NULL</strong>.</td>
</tr>
<tr>
<td></td>
<td>The EFI DHCPv6 Protocol instance has already started the DHCPv6 S.A.R.R</td>
</tr>
<tr>
<td></td>
<td>when <strong>Dhcp6CfgData</strong> is <strong>NULL</strong></td>
</tr>
<tr>
<td>EFI_OUT_OF_RESOURCES</td>
<td>Required system resources could not be allocated.</td>
</tr>
<tr>
<td>EFI_DEVICE_ERROR</td>
<td>An unexpected system or network error occurred.</td>
</tr>
</tbody>
</table>

**EFI_DHCP6_PROTOCOL.Start ()**

Summary
Start the DHCPv6 S.A.R.R process.

Prototype
```c
typedef
    EFI_STATUS
        (EFIAPI *EFI_DHCP6_START) (          
          IN EFI_DHCP6_PROTOCOL  *This       
        );
```

Parameters
**This**
Pointer to the **EFI_DHCP6_PROTOCOL** instance.

Description
The **Start()** function starts the DHCPv6 S.A.R.R process. This function can be called only when the state of the configured IA is in the **Dhcp6Init** state. If the DHCPv6 S.A.R.R process completes successfully, the state of the configured IA will be transferred through **Dhcp6Selecting** and **Dhcp6Requesting** to
Dhcp6Bound state. The update of the IPv6 addresses will be notified through **EFI_DHCP6_CONFIG_DATA.IaInfoEvent**. At the time when each event occurs in this process, the callback function set by **EFI_DHCP6_PROTOCOL.Configure()** will be called and the user can take this opportunity to control the process. If **EFI_DHCP6_CONFIG_DATA.IaInfoEvent** is **NULL**, the **Start()** function call is a blocking operation. It will return after the DHCPv6 S.A.R.R process completes or aborted by users. If the process is aborted by system or network error, the state of the configured IA will be transferred to Dhcp6Init. The **Start()** function can be called again to restart the process.

**Status Codes Returned**

<table>
<thead>
<tr>
<th>Code</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>EFI_SUCCESS</td>
<td>The DHCPv6 S.A.R.R process is completed and at least one IPv6 address has been bound to the configured IA when <strong>EFI_DHCP6_CONFIG_DATA.IaInfoEvent</strong> is <strong>NULL</strong>.</td>
</tr>
<tr>
<td>EFI_ACCESS_DENIED</td>
<td>The EFI DHCPv6 Child instance hasn’t been configured.</td>
</tr>
<tr>
<td>EFI_INVALID_PARAMETER</td>
<td>This is <strong>NULL</strong>.</td>
</tr>
<tr>
<td>EFI_OUT_OF_RESOURCES</td>
<td>Required system resources could not be allocated.</td>
</tr>
<tr>
<td>EFI_ALREADY_STARTED</td>
<td>The DHCPv6 S.A.R.R process has already started.</td>
</tr>
<tr>
<td>EFI_DEVICE_ERROR</td>
<td>An unexpected network or system error occurred.</td>
</tr>
<tr>
<td>EFI_NO_RESPONSE</td>
<td>The DHCPv6 S.A.R.R process failed because of no response.</td>
</tr>
<tr>
<td>EFI_NO_MAPPING</td>
<td>No IPv6 address has been bound to the configured IA after the DHCPv6 S.A.R.R process.</td>
</tr>
<tr>
<td>EFI_ABORTED</td>
<td>The DHCPv6 S.A.R.R process aborted by user.</td>
</tr>
<tr>
<td>EFI_NO_MEDIA</td>
<td>There was a media error.</td>
</tr>
</tbody>
</table>

**EFI_DHCP6_PROTOCOL.InfoRequest ()**

**Summary**

Request configuration information without the assignment of any IA addresses of the client.

**Prototype**

```c
Typedef
EFI_STATUS
(EFI_API *EFI_DHCP6_INFO_REQUEST) (  
  IN EFI_DHCP6_PROTOCOL    *This,
  IN BOOLEAN SendClientId,
  IN EFI_DHCP6_PACKET_OPTION *OptionRequest,
  IN UINT32 OptionCount,
  IN EFI_DHCP6_PACKET_OPTION *OptionList[]   OPTIONAL,
  IN EFI_DHCP6_RETRANSMISSION *Retransmission,
  IN EFI_EVENT TimeoutEvent    OPTIONAL,
  IN EFI_DHCP6_INFO_CALLBACK ReplyCallback,
  IN VOID           *CallbackContext OPTIONAL
);
```
Parameters

This

Pointer to the EFI_DHCP6_PROTOCOL instance.

SendClientId

If TRUE, the EFI DHCPv6 Protocol instance will build Client Identifier option and include it into Information Request packet. If FALSE, Client Identifier option will not be included. Client Identifier option can not be specified through OptionList parameter.

OptionRequest

Pointer to the Option Request option in the Information Request packet. Option Request option can not be specified through OptionList parameter.

OptionCount

Number of options in OptionList.

OptionList

List of other DHCPv6 options. These options will be appended to the Option Request option. The caller is responsible for freeing this buffer. Type is defined in EFI_DHCP6_PROTOCOL.GetModeData().

Retransmission

Parameter to control Information Request packet retransmission behavior. Type EFI_DHCP6_RETRANSMISSION is defined in “Related Definition” below. The buffer can be freed after EFI_DHCP6_PROTOCOL.InfoRequest() returns.

TimeoutEvent

If not NULL, this event is signaled when the information request exchange aborted because of no response. If NULL, the function call is a blocking operation; and it will return after the information-request exchange process finish or aborted by users.

ReplyCallback

The callback function is to intercept various events that occur in the Information Request exchange process. It should not be set to NULL. Type EFI_DHCP6_INFO_CALLBACK is defined below.

CallbackContext

Pointer to the context that will be passed to ReplyCallback.

Description

The InfoRequest() function is used to request configuration information without the assignment of any IPv6 address of the client. Client sends out Information Request packet to obtain the required configuration information, and DHCPv6 server responds with Reply packet containing the information for the client. The received Reply packet will be passed to the user by ReplyCallback function. If user returns EFI_NOT_READY from ReplyCallback, the EFI DHCPv6 Protocol instance will continue to receive other Reply packets unless timeout according to the Retransmission parameter. Otherwise, the Information Request exchange process will be finished successfully if user returns EFI_SUCCESS from ReplyCallback.
Related Definitions

```c
typedef EFI_STATUS (EFIAPI *EFI_DHCP6_INFO_CALLBACK)(
    IN EFI_DHCP6_PROTOCOL  *This,
    IN VOID          *Context,
    IN EFI_DHCP6_PACKET    *Packet,
);
```

*This*  
Pointer to the `EFI_DHCP6_PROTOCOL` instance that is used to configure this callback function.

*Context*  
Pointer to the context that is initialized in the `EFI_DHCP6_PROTOCOL.InfoRequest()`.

*Packet*  
Pointer to Reply packet that has been received. The EFI DHCPv6 Protocol instance is responsible for freeing the buffer. Type `EFI_DHCP6_PACKET` is defined in `EFI_DHCP6_PROTOCOL.GetModeData()`.

`EFI_DHCP6_INFO_CALLBACK` is provided by the consumer of the EFI DHCPv6 Protocol instance to intercept events that occurs in the DHCPv6 Information Request exchange process. There are three possible returned values, which are described in the following table.

<table>
<thead>
<tr>
<th>Value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>EFI_SUCCESS</td>
<td>Tell the EFI DHCPv6 Protocol instance to finish Information Request exchange process.</td>
</tr>
<tr>
<td>EFI_NOT_READY</td>
<td>Tell the EFI DHCPv6 Protocol instance to continue Information Request exchange process.</td>
</tr>
<tr>
<td>EFI_ABORTED</td>
<td>Tell the EFI DHCPv6 Protocol instance to abort the Information Request exchange process</td>
</tr>
</tbody>
</table>
Status Codes Returned

| EFI.SUCCESS | The DHCPv6 information request exchange process completed when TimeoutEvent is NULL. Information Request packet has been sent to DHCPv6 server when TimeoutEvent is not NULL. |
|EFI_INVALID_PARAMETER | One or more following conditions are TRUE:  
• This is NULL.  
• OptionRequest is NULL or OptionRequest->OpCode is invalid.  
• OptionCount > 0 and OptionList is NULL.  
• OptionList is not NULL, and Client Identify option or Option Request option is specified in the OptionList.  
• Retransmission is NULL.  
• Both Retransmission->Mrc and Retransmission->Mrd are zero.  
• ReplyCallback is NULL. |
|EFI DEVICE_ERROR | An unexpected network or system error occurred. |
|EFI_NO_RESPONSE | The DHCPv6 information request exchange process failed because of no response, or not all requested-options are responded by DHCPv6 servers when Timeout happened. |
|EFI_ABORTED | The DHCPv6 information request exchange process aborted by user. |

**EFI_DHCP6_PROTOCOL.RenewRebind ()**

**Summary**

Manually extend the valid and preferred lifetimes for the IPv6 addresses of the configured IA and update other configuration parameters by sending Renew or Rebind packet.

**Prototype**

```c
typedef EFI_STATUS (EFIAPIC *EFI_DHCP6_RENEW_REBIND) (  
    IN EFI_DHCP6_PROTOCOL *This,  
    IN BOOLEAN RebindRequest  
    );
```

**Parameters**

**This**

Pointer to the EFI_DHCP6_PROTOCOL instance.

**RebindRequest**

If TRUE, it will send Rebind packet and enter the Dhcp6Rebinding state. Otherwise, it will send Renew packet and enter the Dhcp6Renewing state.
Description

The RenewRebind() function is used to manually extend the valid and preferred lifetimes for the IPv6 addresses of the configured IA and update other configuration parameters by sending Renew or Rebind packet.

- When RebindRequest is FALSE and the state of the configured IA is Dhcp6Bound, it will send Renew packet to the previously DHCPv6 server and transfer the state of the configured IA to Dhcp6Renewing. If valid Reply packet received, the state transfers to Dhcp6Bound and the valid and preferred timer restarts. If fails, the state transfers to Dhcp6Bound but the timer continues.

- When RebindRequest is TRUE and the state of the configured IA is Dhcp6Bound, it will send Rebind packet. If valid Reply packet received, the state transfers to Dhcp6Bound and the valid and preferred timer restarts. If fails, the state transfers to Dhcp6Init and the IA can’t be used.

Status Codes Returned

<table>
<thead>
<tr>
<th>Code</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>EFI_SUCCESS</td>
<td>The DHCPv6 renew/rebind exchange process has completed and at least one IPv6 address of the configured IA has been bound again when EFI_DHCP6_CONFIG_DATA.IaInfoEvent is NULL. The EFI DHCPv6 Protocol instance has sent Renew or Rebind packet when EFI_DHCP6_CONFIG_DATA.IaInfoEvent is not NULL.</td>
</tr>
<tr>
<td>EFI_ACCESS_DENIED</td>
<td>The EFI DHCPv6 Child instance hasn’t been configured, or the state of the configured IA is not in Dhcp6Bound.</td>
</tr>
<tr>
<td>EFI_ALREADY_STARTED</td>
<td>The state of the configured IA has already entered Dhcp6Renewing when RebindRequest is FALSE. The state of the configured IA has already entered Dhcp6Rebinding when RebindRequest is TRUE.</td>
</tr>
<tr>
<td>EFI_INVALID_PARAMETER</td>
<td>This is NULL.</td>
</tr>
<tr>
<td>EFI_DEVICE_ERROR</td>
<td>An unexpected network or system error occurred.</td>
</tr>
<tr>
<td>EFI_NO_RESPONSE</td>
<td>The DHCPv6 renew/rebind exchange process failed because of no response.</td>
</tr>
<tr>
<td>EFI_NO_MAPPING</td>
<td>No IPv6 address has been bound to the configured IA after the DHCPv6 renew/rebind exchange process.</td>
</tr>
<tr>
<td>EFI_ABORTED</td>
<td>The DHCPv6 renew/rebind exchange process aborted by user.</td>
</tr>
</tbody>
</table>

EFI_DHCP6_PROTOCOL.Decline()

Summary

Inform that one or more IPv6 addresses assigned by a server are already in use by another node.
Prototype

```c
typedef
  EFI_STATUS
  (EFIAPI *EFI_DHCP6_DECLINE) (
    IN EFI_DHCP6_PROTOCOL   *This,
    IN UINT32 AddressCount,
    IN EFI_IPv6_ADDRESS    *Addresses
  );
```

Parameters

- **This**
  - Pointer to the `EFI_DHCP6_PROTOCOL` instance.
- **AddressCount**
  - Number of declining IPv6 addresses.
- **Addresses**
  - Pointer to the buffer stored all the declining IPv6 addresses.

Description

The `Decline()` function is used to manually decline the assignment of IPv6 addresses, which have been already used by another node. If all IPv6 addresses of the configured IA are declined through this function, the state of the IA will switch through `Dhcp6Declining` to `Dhcp6Init`, otherwise, the state of the IA will restore to `Dhcp6Bound` after the declining process. The `Decline()` can only be called when the IA is in `Dhcp6Bound` state. If the `EFI_DHCP6_CONFIG_DATA.IaInfoEvent` is NULL, this function is a blocking operation. It will return after the declining process finishes, or aborted by user.

Status Codes Returned

<table>
<thead>
<tr>
<th>Status Code</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>EFI_SUCCESS</td>
<td>The DHCPv6 decline exchange process has completed when <code>EFI_DHCP6_CONFIG_DATA.IaInfoEvent</code> is NULL.</td>
</tr>
<tr>
<td></td>
<td>The EFI DHCPv6 Protocol instance has sent Decline packet when <code>EFI_DHCP6_CONFIG_DATA.IaInfoEvent</code> is not NULL.</td>
</tr>
<tr>
<td>EFI_INVALID_PARAMETER</td>
<td>One or more following conditions are TRUE.</td>
</tr>
<tr>
<td></td>
<td>• This is NULL.</td>
</tr>
<tr>
<td></td>
<td>• AddressCount is zero or Addresses is NULL.</td>
</tr>
<tr>
<td>EFI_NOT_FOUND</td>
<td>Any specified IPv6 address is not correlated with the configured IA for this instance.</td>
</tr>
<tr>
<td>EFI_ACCESS_DENIED</td>
<td>The EFI DHCPv6 Child instance hasn’t been configured, or the state of the configured IA is not in <code>Dhcp6Bound</code>.</td>
</tr>
<tr>
<td>EFI_DEVICE_ERROR</td>
<td>An unexpected network or system error occurred.</td>
</tr>
<tr>
<td>EFI_ABORTED</td>
<td>The DHCPv6 decline exchange process aborted by user.</td>
</tr>
</tbody>
</table>
**EFI_DHCP6_PROTOCOL.Release ()**

**Summary**
Release one or more IPv6 addresses associated with the configured IA for current instance.

**Prototype**
```
typedef
    EFI_STATUS
    (EFIAPI *EFI_DHCP6_RELEASE) (  
        IN EFI_DHCP6_PROTOCOL   *This,
        IN UINT32 AddressCount,
        IN EFI_IPv6_ADDRESS    *Addresses
    );
```

**Parameters**
- **This**
  Pointer to the ** EFI_DHCP6_PROTOCOL** instance.
- **AddressCount**
  Number of releasing IPv6 addresses.
- **Addresses**
  Pointer to the buffer stored all the releasing IPv6 addresses. Ignored if **AddressCount** is zero.

**Description**
The **Release()** function is used to manually release the one or more IPv6 address. If **AddressCount** is zero, it will release all IPv6 addresses of the configured IA. If all IPv6 addresses of the IA are released through this function, the state of the IA will switch through **Dhcp6Releasing** to **Dhcp6Init**, otherwise, the state of the IA will restore to **Dhcp6Bound** after the releasing process. The **Release()** can only be called when the IA is in **Dhcp6Bound** state. If the **EFI_DHCP6_CONFIG_DATA.IaInfoEvent** is **NULL**, the function is a blocking operation. It will return after the releasing process finishes, or aborted by user.
Status Codes Returned

<table>
<thead>
<tr>
<th>Status Code</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>EFI_SUCCESS</td>
<td>The DHCPv6 release exchange process has completed when <code>EFI_DHCP6_CONFIG_DATA.IaInfoEvent</code> is NULL. The EFI DHCPv6 Protocol instance has sent Release packet when <code>EFI_DHCP6_CONFIG_DATA.IaInfoEvent</code> is not NULL.</td>
</tr>
</tbody>
</table>
| EFI_INVALID_PARAMETER | One or more following conditions are TRUE:  
  - `This` is `NULL`.  
  - `AddressCount` is not zero and `Addresses` is `NULL`. |
| EFI_NOT_FOUND | Any specified IPv6 address is not correlated with the configured IA for this instance. |
| EFI_ACCESS_DENIED | The EFI DHCPv6 Child instance hasn’t been configured, or the state of the configured IA is not in `Dhcp6Bound`. |
| EFI_DEVICE_ERROR | An unexpected network or system error occurred. |
| EFI_ABORTED | The DHCPv6 release exchange process aborted by user. |

`EFI_DHCP6_PROTOCOL.Stop()`

**Summary**

Stop the DHCPv6 S.A.R.R process.

**Prototype**

```c
typedef
EFI_STATUS
(EFIAPI *EFI_DHCP6_STOP) (  
    IN EFI_DHCP6_PROTOCOL *This  
);
```

**Parameters**

`This`

Pointer to the `EFI_DHCP6_PROTOCOL` instance.

**Description**

The `Stop()` function is used to stop the DHCPv6 S.A.R.R process. If this function is called successfully, all the IPv6 addresses of the configured IA will be released and the state of the configured IA will be transferred to `Dhcp6Init`. 
Status Codes Returned

<table>
<thead>
<tr>
<th>Code</th>
<th>Message</th>
</tr>
</thead>
<tbody>
<tr>
<td>EFI_SUCCESS</td>
<td>The DHCPv6 S.A.R.R process has been stopped when __EFI_DHCP6_CONFIG_DATA.IaInfoEvent__ is NULL. The EFI DHCPv6 Protocol instance has sent Release packet if need release or has been stopped if needn't, when __EFI_DHCP6_CONFIG_DATA.IaInfoEvent__ is not NULL.</td>
</tr>
<tr>
<td>EFI_INVALID_PARAMETER</td>
<td>This is NULL.</td>
</tr>
<tr>
<td>EFI_NO_MEDIA</td>
<td>There was a media error.</td>
</tr>
</tbody>
</table>

**EFI_DHCP6_PROTOCOL.Parse ()**

**Summary**

Parse the option data in the DHCPv6 packet.

**Prototype**

```c
typedef EFI_STATUS
  (EFIAPI *EFI_DHCP6_PARSE) (
    IN EFI_DHCP6_PROTOCOL *This,
    IN EFI_DHCP6_PACKET *Packet,
    IN OUT UINT32 *OptionCount,
    IN EFI_DHCP6_PACKET_OPTION *PacketOptionList[] OPTIONAL
  );
```

**Parameters**

- **This**
  Pointer to the EFI_DHCP6_PROTOCOL instance.

- **Packet**
  Pointer to packet to be parsed. Type EFI_DHCP6_PACKET is defined in EFIisodes.IDE_DHCP6_PROTOCOL.GetModeData().

- **OptionCount**
  On input, the number of entries in the PacketOptionList. On output, the number of DHCPv6 options in the Packet.

- **PacketOptionList**
  List of pointers to the DHCPv6 options in the Packet. Type EFI_DHCP6_PACKET_OPTION is defined in EFI_DHCP6_PROTOCOL.Configure(). The OpCode and OpLen in EFI_DHCP6_PACKET_OPTION are both stored in network byte order.

**Description**

The Parse() function is used to retrieve the option list in the DHCPv6 packet.
29.4 EFI DNSv4 Protocol

This section defines the EFI Domain Name Service Binding Protocol interface. It is split into the following two main sections.

- DNSv4 Service Binding Protocol (DNSv4SB)
- DNSv4 Protocol (DNSv4)

**EFI_DNS4_SERVICE_BINDING_PROTOCOL**

**Summary**

The DNSv4SB is used to locate communication devices that are supported by a DNS driver and to create and destroy instances of the DNS child protocol driver.

The EFI Service Binding Protocol in Section 2.5.8 defines the generic Service Binding Protocol functions. This section discusses the details that are specific to the DNSv4.

**GUID**

```plaintext
#define EFI_DNS4_SERVICE_BINDING_PROTOCOL_GUID \
{ 0xb625b186, 0xe063, 0x44f7,\ 
{ 0x89, 0x5, 0x6a, 0x74, 0xdc, 0x6f, 0x52, 0xb4}}
```

**Description**

A network application (or driver) that requires network address resolution can use one of the protocol handler services, such as `BS->LocateHandleBuffer()`, to search for devices that publish a DNSv4SB GUID. Each device with a published DNSv4SB GUID supports DNS and may be available for use.

After a successful call to the `EFI_DNS4_SERVICE_BINDING_PROTOCOL.CreateChild()` function, the child DNS driver instance is in an unconfigured state; it is not ready to resolve addresses.

Before a network application terminates execution, every successful call to the `EFI_DNS4_SERVICE_BINDING_PROTOCOL.CreateChild()` function must be matched with a call to the `EFI_DNS4_SERVICE_BINDING_PROTOCOL.DestroyChild()` function.

<table>
<thead>
<tr>
<th>Status Code</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>EFI_SUCCESS</td>
<td>The packet was successfully parsed.</td>
</tr>
<tr>
<td>EFI_INVALID_PARAMETER</td>
<td>One or more following conditions are <strong>TRUE</strong></td>
</tr>
<tr>
<td></td>
<td>- <strong>This</strong> is <strong>NULL</strong>.</td>
</tr>
<tr>
<td></td>
<td>- <strong>Packet</strong> is <strong>NULL</strong>.</td>
</tr>
<tr>
<td></td>
<td>- <strong>Packet</strong> is not a well-formed DHCPv6 packet.</td>
</tr>
<tr>
<td></td>
<td>- <strong>OptionCount</strong> is <strong>NULL</strong>.</td>
</tr>
<tr>
<td></td>
<td>- <em>OptionCount</em> is not zero and <strong>PacketOptionList</strong> is <strong>NULL</strong>.</td>
</tr>
</tbody>
</table>
| EFI_BUFFER_TOO_SMALL        | *OptionCount* is smaller than the number of options that were found in the **Packet**.
Note: All the network addresses that are described in EFI_DNS4_PROTOCOL are stored in network byte order. Both incoming and outgoing DNS packets are also in network byte order. All other parameters that are defined in functions or data structures are stored in host byte order.

EFI_DNS4_PROTOCOL

Summary

This protocol provides the function to get the host name and address mapping, also provides pass through interface to retrieve arbitrary information from DNS.

The EFI_DNS4_Protocol is primarily intended to retrieve host addresses using the standard DNS protocol (RFC1035), and support for this protocol is required. Implementations may optionally also support local network name resolution methods such as LLMNR (RFC4795) however DNS queries shall always take precedence, and any use of local network name protocols would be restricted to cases where resolution using DNS protocol fails.

As stated above, all instances of EFI_DNS4_Protocol will utilize a common DNS cache containing the successful results of previous queries on any interface. However, it should be noted that every instance of EFI_DNS4_Protocol is associated with a specific network device or interface, and that all network actions initiated using a specific instance of the DNS protocol will occur only via use of the associated network interface. This means, in a system with multiple network interfaces, that a specific DNS server will often only be reachable using a specific network instance, and therefore the protocol user will need to take steps to insure the DNS instance associated with the proper network interface is used. Or alternatively, the caller may perform DNS functions against all interfaces until successful result is achieved.

GUID

```
#define EFI_DNS4_PROTOCOL_GUID \
{ 0xae3d28cc, 0xe05b, 0x4fa1,\ 
  {0xa0, 0x11, 0x7e, 0xb5, 0x5a, 0x3f, 0x14, 0x1 }}
```

Protocol Interface Structure

```
typedef struct _EFI_DNS4_PROTOCOL {
  EFI_DNS4_GET_MODE_DATA       GetModeData;
  EFI_DNS4_CONFIGURE           Configure;
  EFI_DNS4_HOST_NAME_TO_IP     HostNameToIp;
  EFI_DNS4_IP_TO_HOST_NAME    IpToHostName;
  EFI_DNS4_GENERAL_LOOKUP     GeneralLookUp;
  EFI_DNS4_UPDATE_DNS_CACHE  UpdateDnsCache;
  EFI_DNS4_POLL               Poll;
  EFI_DNS4_CANCEL            Cancel;
} EFI_DNS4_PROTOCOL;
```
**EFI_DNS4_PROTOCOL.GetModeData()**

**Summary**
Retrieve the current mode data of this DNS instance.

**Prototype**
```c
typedef
EFI_STATUS
(EIFIAPI *EFI_DNS4_GET_MODE_DATA)(
    IN EFI_DNS4_PROTOCOL     *This,
    OUT EFI_DNS4_MODE_DATA    *DnsModeData
);
```

**Description**
This function is used to retrieve DNS mode data for this DNS instance.

**Parameter**
- **This** Pointer to `EFI_DNS4_PROTOCOL` instance.
- **DnsModeData** Pointer to the caller-allocated storage for the `EFI_DNS4_MODE_DATA` structure.

**Related Definitions**
```c
//**********************************************
// EFI_DNS4_MODE_DATA
//**********************************************
typedef struct {
    EFI_DNS4_CONFIG_DATA DnsConfigData;
    UINT32 DnsServerCount;
    EFI_IPv4_ADDRESS   *DnsServerList;
    UINT32 DnsCacheCount;
    EFI_DNS4_CACHE_ENTRY *DnsCacheList;
} EFI_DNS4_MODE_DATA;
```

- **DnsConfigData** The current configuration data of this instance. Type `EFI_DNS4_CONFIG_DATA` is defined below.
- **DnsServerCount** Number of configured DNS servers.
- **DnsServerList** Pointer to common list of addresses of all configured DNS server used by `EFI_DNS4_PROTOCOL` instances. List will include DNS servers configured by this or any other `EFI_DNS4_PROTOCOL` instance. The storage for this list is allocated by the driver publishing this protocol, and must be freed by the caller.
- **DnsCacheCount** Number of DNS Cache entries. The DNS Cache is shared among all DNS instances.
- **DnsCacheList** Pointer to a buffer containing `DnsCacheCount` DNS Cache entry structures. The storage for this list is allocated by the driver publishing this protocol and must be freed by caller.
typedef struct {
  UINTN  DnsServerListCount;
  EFI_IPv4_ADDRESS  *DnsServerList;
  BOOLEAN       UseDefaultSetting;
  BOOLEAN       EnableDnsCache;
  UINT8          Protocol;
  EFI_IPv4_ADDRESS  StationIp;
  EFI_IPv4_ADDRESS  SubnetMask;
  UINT16         LocalPort;
  UINT32         RetryCount;
  UINT32         RetryInterval;
} EFI_DNS4_CONFIG_DATA;

DnsServerListCount  Count of the DNS servers. When used with GetModeData(), this field is the count of originally configured servers when Configure() was called for this instance. When used with Configure() this is the count of caller-supplied servers. If the DnsServerListCount is zero, the DNS server configuration will be retrieved from DHCP server automatically.

DnsServerList  Pointer to DNS server list containing DnsServerListCount entries or NULL if DnsServerListCount is 0. For Configure(), this will be NULL when there are no caller-supplied server addresses, and, the DNS instance will retrieve DNS server from DHCP Server. The provided DNS server list is recommended to be filled up in the sequence of preference. When used with GetModeData(), the buffer containing the list will be allocated by the driver implementing this protocol and must be freed by the caller. When used with Configure(), the buffer containing the list will be allocated and released by the caller.

UseDefaultSetting  Set to TRUE to use the default IP address/subnet mask and default routing table.

EnableDnsCache  If TRUE, enable DNS cache function for this DNS instance. If FALSE, all DNS query will not lookup local DNS cache.

Protocol  Use the protocol number defined in “Links to UEFI-Related Documents” (http://uefi.org/uefi) under the heading “IANA Protocol Numbers”. Only TCP or UDP are supported, and other protocol values are invalid. An implementation can choose to support only UDP, or both TCP and UDP.

StationIp  If UseDefaultSetting is FALSE indicates the station address to use.

SubnetMask  If UseDefaultSetting is FALSE indicates the subnet mask to use.

LocalPort  Local port number. Set to zero to use the automatically assigned port number.

RetryCount  Retry number if no response received after RetryInterval.
RetryInterval

Minimum interval of retry is 2 second. If the retry interval is less than 2 second, then use the 2 second.

//***************************************************************
// EFI_DNS4_CACHE_ENTRY //***************************************************************
typedef struct {
    CHAR16    *HostName;
    EFI_IPv4_ADDRESS  *IpAddress;
    UINT32  Timeout;
} EFI_DNS4_CACHE_ENTRY;

HostName

Host name.

IpAddress

IP address of this host.

Timeout

Time in second unit that this entry will remain in DNS cache. A value of zero means that this entry is permanent. A nonzero value will override the existing one if this entry to be added is dynamic entry. Implementations may set its default timeout value for the dynamically created DNS cache entry after one DNS resolve succeeds.

Status Codes Returned

<table>
<thead>
<tr>
<th>Code</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>EFI_SUCCESS</td>
<td>The operation completed successfully.</td>
</tr>
<tr>
<td>EFI_NOT_STARTED</td>
<td>When DnsConfigData is queried, no configuration data is available because this instance has not been configured.</td>
</tr>
<tr>
<td>EFI_INVALID_PARAMETER</td>
<td>This is NULL or DnsModeData is NULL.</td>
</tr>
<tr>
<td>EFI_OUT_OF_RESOURCES</td>
<td>Failed to allocate needed resources.</td>
</tr>
</tbody>
</table>

EFI_DNS4_PROTOCOL.Configure()

Summary

Configures this DNS instance.

Prototype

typedef
    EFI_STATUS
    (EFIAPI *EFI_DNS4_CONFIGURE)(
        IN EFI_DNS4_PROTOCOL      *This,
        IN EFI_DNS4_CONFIG_DATA    *DnsConfigData
    );

Descriptions

This function is used to configure DNS mode data for this DNS instance.
Parameters

- **This**: Pointer to **EFI_DNS4_PROTOCOL** instance.
- **DnsConfigData**: Pointer to caller-allocated buffer containing **EFI_DNS4_CONFIG_DATA** structure containing the desired Configuration data. If NULL, the driver will reinitialize the protocol instance to the unconfigured state.

Status Codes Returned

<table>
<thead>
<tr>
<th>Code</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>EFI_SUCCESS</td>
<td>The operation completed successfully.</td>
</tr>
<tr>
<td>EFI_UNSUPPORTED</td>
<td>The designated protocol is not supported.</td>
</tr>
<tr>
<td>EFI_INVALID_PARAMETER</td>
<td><strong>This</strong> is NULL. The StationIp address provided in <strong>DnsConfigData</strong> is not a valid unicast. The <strong>DnsServerList</strong> is NULL while <strong>DnsServerListCount</strong> is not zero. <strong>DnsServerListCount</strong> is zero while <strong>DnsServerList</strong> is not NULL.</td>
</tr>
<tr>
<td>EFI_OUT_OF_RESOURCES</td>
<td>The DNS instance data or required space could not be allocated.</td>
</tr>
<tr>
<td>EFI_DEVICE_ERROR</td>
<td>An unexpected system or network error occurred. The EFI DNSv4 Protocol instance is not configured.</td>
</tr>
<tr>
<td>EFI_ALREADY_STARTED</td>
<td>Second call to <strong>Configure()</strong> with <strong>DnsConfigData</strong>. To reconfigure the instance the caller must call <strong>Configure()</strong> with NULL first to return driver to unconfigured state.</td>
</tr>
</tbody>
</table>

**EFI_DNS4_PROTOCOL.HostNameToIp()**

Summary

- Host name to host address translation.

Prototype

```c
typedef
  EFI_STATUS
(EFI_API *EFI_DNS4_HOST_NAME_TO_IP) (  
    IN EFI_DNS4_PROTOCOL  *This,
    IN CHAR16            *HostName,
    IN EFI_DNS4_COMPLETION_TOKEN   *Token
  );
```

Parameter

- **This**: Pointer to **EFI_DNS4_PROTOCOL** instance.
- **Hostname**: Pointer to buffer containing fully-qualified Domain Name including Hostname. To resolve successfully, characters within the FQDN string must be chosen according to the format and from within the set of ASCII characters authorized by DNS specifications. Any translation required for reference to domains or hostnames defined...
as containing Unicode characters, for example use of Punycode, must be performed by caller.

**Token**

Pointer to the caller-allocated completion token to return at the completion of the process to translate host name to host address. Type **EFI_DNS4_COMPLETION_TOKEN** is defined in "Related Definitions" below.

**Related Definition**

```c
typedef struct {
    EFI_EVENT      Event;
    EFI_STATUS      Status;
    UINT32        RetryCount;
    UINT32        RetryInterval;
    union {
        DNS_HOST_TO_ADDR_DATA    *H2AData;
        DNS_ADDR_TO_HOST_DATA    *A2HData;
        DNS_GENERAL_LOOKUP_DATA   *GLookupData;
    } RspData;
} EFI_DNS4_COMPLETION_TOKEN;
```

**Event**

This **Event** will be signaled after the **Status** field is updated by the EFI DNS protocol driver. The type of **Event** must be **EFI_NOTIFY_SIGNAL**.

**Status**

Will be set to one of the following values.

- **EFI_SUCCESS**: The host name to address translation completed successfully.
- **EFI_NOT_FOUND**: No matching Resource Record (RR) is found.
- **EFI_TIMEOUT**: No DNS server reachable, or RetryCount was exhausted without response from all specified DNS servers.
- **EFI_DEVICE_ERROR**: An unexpected system or network error occurred.
- **EFI_NO_MEDIA**: There was a media error.

**RetryCount**

Retry number if no response received after **RetryInterval**. If zero, use the parameter configured through Dns.Configure() interface.

**RetryInterval**

Minimum interval of retry is 2 second. If the retry interval is less than 2 second, then use the 2 second. If zero, use the parameter configured through Dns.Configure() interface.

**H2AData**

When the Token is used for host name to address translation, **H2AData** is a pointer to the **DNS_HOST_TO_ADDR_DATA**. Type **DNS_HOST_TO_ADDR_DATA** is defined below.

**A2HData**

When the Token is used for host address to host name translation, **A2HData** is a pointer to the **DNS_ADDR_TO_HOST_DATA**. Type **DNS_ADDR_TO_HOST_DATA** is defined below.
When the Token is used for a general lookup function, GLookupDATA is a pointer to the DNS GENERAL_LOOKUP_DATA. Type DNSGENERAL_LOOKUP_DATA is defined below.

EFI DNS4_COMPLETION_TOKEN structures are used for host name to address translation, host address to name translation and general lookup operation, the Event, RetryCount and RetryInterval fields filled must be filled by the EFI DNS4 Protocol Client. After the operation completes, the EFI DNS4 protocol driver fill in the RspData and Status field and the Event is signaled.

```c
//******************************************
// DNS_HOST_TO_ADDR_DATA
//******************************************
typedef struct {
    UINT32           IpCount;
    EFI_IPv4_Address      *IpList;
} DNS_HOST_TO_ADDR_DATA;

IpCount Number of the returned IP addresses.
IpList Pointer to the all the returned IP addresses.

//******************************************
// DNS_ADDR_TO_HOST_DATA
//******************************************
typedef struct {
    CHAR16            *HostName;
} DNS_ADDR_TO_HOST_DATA;

HostName Pointer to the primary name for this host address. It’s the caller’s responsibility to free the response memory.

//******************************************
// DNS_GENERAL_LOOKUP_DATA
//******************************************
typedef struct {
    UINTN
    RRCount;
    DNS_RESOURCE_RECORD  *RRList;
} DNS_GENERAL_LOOKUP_DATA;

RRCount Number of returned matching RRs.
RRList Pointer to the all the returned matching RRs. It’s caller responsibility to free the allocated memory to hold the returned RRs.
typedef struct {
    CHAR8    *QName;
    UINT16   QType;
    UINT16   QClass;
    UINT32   TTL;
    UINT16   DataLength;
    CHAR8    *RData;
} DNS_RESOURCE_RECORD;

QName        The Owner name.
QType        The Type Code of this RR.
QClass       The CLASS code of this RR.
TTL          32 bit integer which specify the time interval that the resource
             record may be cached before the source of the information should
             again be consulted. Zero means this RR cannot be cached.
DataLength   16 big integer which specify the length of RData.
RData        A string of octets that describe the resource, the format of this
             information varies according to QType and QClass difference.

Description

The HostNameToIp() function is used to translate the host name to host IP address. A
type A query is used to get the one or more IP addresses for this host.

Status Codes Returned

<table>
<thead>
<tr>
<th>Status Code</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>EFI_SUCCESS</td>
<td>The operation was queued successfully.</td>
</tr>
<tr>
<td>EFI_INVALID_PARAMETER</td>
<td>One or more of the following conditions is TRUE</td>
</tr>
<tr>
<td></td>
<td>This is NULL.</td>
</tr>
<tr>
<td></td>
<td>Token is NULL.</td>
</tr>
<tr>
<td></td>
<td>Token.Event is NULL</td>
</tr>
<tr>
<td></td>
<td>HostName is NULL.</td>
</tr>
<tr>
<td></td>
<td>HostName string is unsupported format.</td>
</tr>
<tr>
<td>EFI_NO_MAPPING</td>
<td>There’s no source address is available for use.</td>
</tr>
<tr>
<td>EFI_NOT_STARTED</td>
<td>This instance has not been started.</td>
</tr>
</tbody>
</table>

EFI_DNS4_PROTOCOL.IpToHostName()

Summary
IPv4 address to host name translation also known as Reverse DNS lookup.
Prototype

```c
typedef EFI_STATUS
(EFI_API *EFI_DNS4_IP_TO_HOST_NAME) (IN EFI_DNS4_PROTOCOL *This,
  IN EFI_IPv4_ADDRESS IpAddress,
  IN EFI_DNS4_COMPLETION_TOKEN *Token);
```

Parameter

- **This**: Pointer to `EFI_DNS4_PROTOCOL` instance.
- **IpAddress**: IP address.
- **Token**: Pointer to the caller-allocated completion used token to translate host address to host name. Type `EFI_DNS4_COMPLETION_TOKEN` is defined in "Related Definitions" of above `HostNameToIp()`.

Description

The `IpToHostName()` function is used to translate the host address to host name. A type PTR query is used to get the primary name of the host. Support of this function is optional.

Status Codes Returned

<table>
<thead>
<tr>
<th>Status Code</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>EFI_SUCCESS</td>
<td>The operation was queued successfully.</td>
</tr>
<tr>
<td>EFI_UNSUPPORTED</td>
<td>This function is not supported</td>
</tr>
<tr>
<td>EFI_INVALID_PARAMETER</td>
<td>One or more of the following conditions is TRUE: This is NULL. Token is NULL. Token.Event is NULL. IpAddress is not valid IP address.</td>
</tr>
<tr>
<td>EFI_NO_MAPPING</td>
<td>There's no source address is available for use.</td>
</tr>
<tr>
<td>EFI_ALREADY_STARTED</td>
<td>This Token is being used in another DNS session.</td>
</tr>
<tr>
<td>EFI_OUT_OF_RESOURCES</td>
<td>Failed to allocate needed resources.</td>
</tr>
</tbody>
</table>

`EFI_DNS4_PROTOCOL.GeneralLookUp()`

Summary

Retrieve arbitrary information from the DNS server.
Prototype

```c
typedef
  EFI_STATUS
  (EFI_API *EFI_DNS4_GENERAL_LOOKUP) (
    IN EFI_DNS4_PROTOCOL        *This,
    IN CHAR8              *QName,
    IN UINT16              QType,
    IN UINT16              QClass,
    IN EFI_DNS4_COMPLETION_TOKEN    *Token
  );
```

Description

This `GeneralLookUp()` function retrieves arbitrary information from the DNS. The caller supplies a `QNAME`, `QTYPE`, and `QCLASS`, and all of the matching RRs are returned. All RR content (e.g., TTL) was returned. The caller need parse the returned RR to get required information. This function is optional.

Parameters

- This: Pointer to `EFI_DNS4_PROTOCOL` instance.
- QName: Pointer to Query Name.
- QType: Query Type.
- QClass: Query Name.
- Token: Pointer to the caller-allocated completion token to retrieve arbitrary information. Type `EFI_DNS4_COMPLETION_TOKEN` is defined in "Related Definitions" of above `HostNameToIp()`.

Status Codes Returned

<table>
<thead>
<tr>
<th>Code</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>EFI_SUCCESS</td>
<td>The operation was queued successfully.</td>
</tr>
<tr>
<td>EFI_UNSUPPORTED</td>
<td>This function is not supported. Or the requested <code>QType</code> is not supported</td>
</tr>
<tr>
<td>EFI_INVALID_PARAMETER</td>
<td>One or more of the following conditions is TRUE</td>
</tr>
<tr>
<td></td>
<td><code>This</code> is <code>NULL</code>.</td>
</tr>
<tr>
<td></td>
<td><code>Token</code> is <code>NULL</code>.</td>
</tr>
<tr>
<td></td>
<td><code>Token.Event</code> is <code>NULL</code>.</td>
</tr>
<tr>
<td></td>
<td><code>QName</code> is <code>NULL</code>.</td>
</tr>
<tr>
<td>EFI_NO_MAPPING</td>
<td>There's no source address is available for use.</td>
</tr>
<tr>
<td>EFI_ALREADY_STARTED</td>
<td>This <code>Token</code> is being used in another DNS session.</td>
</tr>
<tr>
<td>EFI_OUT_OF_RESOURCES</td>
<td>Failed to allocate needed resources.</td>
</tr>
</tbody>
</table>

`EFI_DNS4_PROTOCOL.UpdateDnsCache()`

Summary

This function is used to update the DNS Cache.
Prototype

```c
typedef
 EFI_STATUS
 (EFIAPI *EFI_DNS4_UPDATE_DNS_CACHE) (  
   IN EFI_DNS4_PROTOCOL   *This,
   IN BOOLEAN            DeleteFlag,
   IN BOOLEAN            Override,
   IN EFI_DNS4_CACHE_ENTRY DnsCacheEntry
  );
```

Parameters

- **This**: Pointer to `EFI_DNS4_PROTOCOL` instance.
- **DeleteFlag**: If `FALSE`, this function is to add one entry to the DNS Cache. If `TRUE`, this function will delete matching DNS Cache entry.
- **Override**: If `TRUE`, the matching DNS cache entry will be overwritten with the supplied parameter. If `FALSE`, `EFI_ACCESS_DENIED` will be returned if the entry to be added is already exists.
- **DnsCacheEntry**: Pointer to DNS Cache entry.

Description

The `UpdateDnsCache()` function is used to add/delete/modify DNS cache entry. DNS cache can be normally dynamically updated after the DNS resolve succeeds. This function provided capability to manually add/delete/modify the DNS cache.

Status Codes Returned

<table>
<thead>
<tr>
<th>Status Code</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>EFI_SUCCESS</code></td>
<td>The operation completed successfully.</td>
</tr>
<tr>
<td><code>EFI_INVALID_PARAMETER</code></td>
<td>One or more of the following conditions is true:</td>
</tr>
<tr>
<td><code>This</code> is <code>NULL</code></td>
<td></td>
</tr>
<tr>
<td><code>DnsCacheEntry.HostName</code> is <code>NULL</code></td>
<td></td>
</tr>
<tr>
<td><code>DnsCacheEntry.IpAddress</code> is <code>NULL</code></td>
<td></td>
</tr>
<tr>
<td><code>DnsCacheEntry.Timeout</code> is zero</td>
<td></td>
</tr>
<tr>
<td><code>EFI_ACCESS_DENIED</code></td>
<td>The DNS cache entry already exists and <code>Override</code> is not <code>TRUE</code>.</td>
</tr>
</tbody>
</table>

`EFI_DNS4_PROTOCOL.Poll()`

Summary

Polls for incoming data packets and processes outgoing data packets.
Prototype

```c
typedef
    EFI_STATUS
    (EFIAPI *EFI_DNS4_POLL) (
        IN EFI_DNS4_PROTOCOL *This
    );
```

Parameters

This Pointer to **EFI_DNS4_PROTOCOL** instance.

Description

The `Poll()` function can be used by network drivers and applications to increase the rate that data packets are moved between the communications device and the transmit and receive queues.

In some systems, the periodic timer event in the managed network driver may not poll the underlying communications device fast enough to transmit and/or receive all data packets without missing incoming packets or dropping outgoing packets. Drivers and applications that are experiencing packet loss should try calling the `Poll()` function more often.

Status Codes Returned

<table>
<thead>
<tr>
<th>Code</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>EFI_SUCCESS</td>
<td>Incoming or outgoing data was processed.</td>
</tr>
<tr>
<td>EFI_NOT_STARTED</td>
<td>This EFI DNS Protocol instance has not been started.</td>
</tr>
<tr>
<td>EFI_INVALID_PARAMETER</td>
<td>This is <strong>NULL</strong>.</td>
</tr>
<tr>
<td>EFI_DEVICE_ERROR</td>
<td>An unexpected system or network error occurred.</td>
</tr>
<tr>
<td>EFI_TIMEOUT</td>
<td>Data was dropped out of the transmit and/or receive queue. Consider increasing the polling rate.</td>
</tr>
</tbody>
</table>

**EFI_DNS4_PROTOCOL.Cancel()**

Summary

Abort an asynchronous DNS operation, including translation between IP and Host, and general look up behavior.

Prototype

```c
EFI Protocol
typedef
    EFI_STATUS
    (EFIAPI *EFI_DNS4_CANCEL) (
        IN EFI_DNS4_PROTOCOL *This,
        IN EFI_DNS4_COMPLETION_TOKEN *Token
    );
```

Parameters

This Pointer to **EFI_DNS4_PROTOCOL** instance.
Token

Pointer to a token that has been issued by
EFI_DNS4_PROTOCOL.HostNameToIp(),
EFI_DNS4_PROTOCOL.IpToHostName() or
EFI_DNS4_PROTOCOL.GeneralLookUp(). If NULL, all pending
tokens are aborted.

Description

The Cancel() function is used to abort a pending resolution request. After calling this
function, Token.Status will be set to EFI_ABORTED and then Token.Event will be
signaled. If the token is not in one of the queues, which usually means that the
asynchronous operation has completed, this function will not signal the token and
EFI_NOT_FOUND is returned.

Status Codes Returned

<table>
<thead>
<tr>
<th>Code</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>EFI_SUCCESS</td>
<td>The asynchronous DNS operation was aborted and Token-&gt;Event is signaled.</td>
</tr>
<tr>
<td>EFI_NOT_STARTED</td>
<td>This EFI DNS4 Protocol instance has not been started.</td>
</tr>
<tr>
<td>EFI_INVALID_PARAMETER</td>
<td>This is NULL.</td>
</tr>
<tr>
<td>EFI_NOT_FOUND</td>
<td>When Token is not NULL, and the asynchronous DNS operation was not found in the transmit queue. It was either completed or was not issued by HostNameToIp(), IpToHostName() or GeneralLookUp().</td>
</tr>
</tbody>
</table>

29.5 EFI DNSv6 Protocol

This section defines the EFI DNSv6 (Domain Name Service version 6) Protocol. It is split into the following
two main sections.

- DNSv6 Service Binding Protocol (DNSv6SB)
- DNSv6 Protocol (DNSv6)

29.5.1 DNSv6 Service Binding Protocol

EFI_DNS6_SERVICE_BINDING_PROTOCOL

Summary

The DNSv6SB is used to locate communication devices that are supported by a DNS
driver and to create and destroy instances of the DNS child protocol driver.

The EFI Service Binding Protocol in Section 2.5.8 defines the generic Service Binding
Protocol functions. This section discusses the details that are specific to the DNSv6.

GUID
#define EFI_DNS6_SERVICE_BINDING_PROTOCOL_GUID \
{ 0x7f1647c8, 0xb76e, 0x44b2,\} 
{ 0xa5, 0x65, 0xf7, 0xf, 0xf1, 0x9c, 0xd1, 0x9e}}

Description

A network application (or driver) that requires network address resolution can use one of the protocol handler services, such as `BS->LocateHandleBuffer()`, to search for devices that publish a DNSv6SB GUID. Each device with a published DNSv6SB GUID supports DNSv6 and may be available for use.

After a successful call to the `EFI_DNS6_SERVICE_BINDING_PROTOCOL.CreateChild()` function, the child DNS driver instance is in an un-configured state; it is not ready to resolve addresses.

Before a network application terminates execution, every successful call to the `EFI_DNS6_SERVICE_BINDING_PROTOCOL.CreateChild()` function must be matched with a call to the `EFI_DNS6_SERVICE_BINDING_PROTOCOL_DestroyChild()` function.

Note: All the network addresses that are described in `EFI_DNS6_PROTOCOL` are stored in network byte order. Both incoming and outgoing DNS packets are also in network byte order. All other parameters that are defined in functions or data structures are stored in host byte order.

29.5.2 DNS6 Protocol

**EFI_DNS6_PROTOCOL**

Summary

This protocol provides the function to get the host name and address mapping, also provide pass through interface to retrieve arbitrary information from DNSv6.

The EFI_DNS6_Protocol is primarily intended to retrieve host addresses using the standard DNS protocol (RFC3596), and support for this protocol is required. Implementations may optionally also support local network name resolution methods such as LLMNR (RFC4795) however DNS queries shall always take precedence, and any use of local network name protocols would be restricted to cases where resolution using DNS protocol fails.

As stated above, all instances of EFI_DNS6_Protocol will utilize a common DNS cache containing the successful results of previous queries on any interface. However, it should be noted that every instance of EFI_DNS6_Protocol is associated with a specific network device or interface, and that all network actions initiated using a specific instance of the DNS protocol will occur only via use of the associated network interface. This means, in a system with multiple network interfaces, that a specific DNS server will often only be reachable using a specific network instance, and therefore the protocol user will need to take steps to insure the DNS instance associated with the proper network interface is used. Or alternatively, the caller may perform DNS functions against all interfaces until successful result is achieved.
GUID

```c
#define EFI_DNS6_PROTOCOL_GUID \
{ 0xca37bc1f, 0xa327, 0x4ae9,\ 
  { 0x82, 0x8a, 0x8c, 0x40, 0xd8, 0x50, 0x6a, 0x17 }}
```

Protocol Interface Structure

```c
typedef struct _EFI_DNS6_PROTOCOL {
  EFI_DNS6_GET_MODE_DATA       GetModeData;
  EFI_DNS6_CONFIGURE           Configure;
  EFI_DNS6_HOST_NAME_TO_IP     HostNameToIp;
  EFI_DNS6_IP_TO_HOST_NAME    IpToHostName;
  EFI_DNS6_GENERAL_LOOKUP     GeneralLookUp;
  EFI_DNS6_UPDATE_DNS_CACHE   UpdateDnsCache;
  EFI_DNS6_POLL               Poll;
  EFI_DNS6_CANCEL             Cancel;
} EFI_DNS6_PROTOCOL;
```

`EFI_DNS6_PROTOCOL.GetModeData()`

**Summary**
Retrieve mode data of this DNS instance.

**Prototype**

```c
typedef
EFI_STATUS
(EIFIAPI *EFI_DNS6_GET_MODE_DATA)(
  IN EFI_DNS6_PROTOCOL      *This,
  OUT EFI_DNS6_MODE_DATA    *DnsModeData
);
```

**Description**
This function is used to retrieve DNS mode data for this DNS instance.

**Parameter**

- **This** Pointer to `EFI_DNS6_PROTOCOL` instance.
- **DnsModeData** Pointer to the caller-allocated storage for the `EFI_DNS6_MODE_DATA` data.
Related Definitions

```c
// EDF_DNS6_MODE_DATA
typedef struct {
  EFI_DNS6_CONFIG_DATA DnsConfigData;
  UINT32 DnsServerCount;
  EFI_IPV6_ADDRESS *DnsServerList;
  UINT32 DnsCacheCount;
  EFI_DNS6_CACHE_ENTRY *DnsCacheList;
} EFI_DNS6_MODE_DATA;
```

- **DnsConfigData** is the configuration data of this instance. Type `EFI_DNS6_CONFIG_DATA` is defined below.
- **DnsServerCount** is the number of configured DNS6 servers.
- **DnsServerList** is a pointer to a common list of addresses of all configured DNS server used by `EFI_DNS6_PROTOCOL` instances. List will include DNS servers configured by this or any other `EFI_DNS6_PROTOCOL` instance. The storage for this list is allocated by the driver publishing this protocol, and must be freed by the caller.
- **DnsCacheCount** is the number of DNS Cache entries. The DNS Cache is shared among all DNS6 instances.
- **DnsCacheList** is a pointer to a buffer containing `DnsCacheCount` DNS Cache entry structures. The storage for this list is allocated by the driver publishing this protocol and must be freed by caller.

```c
// EFI_DNS6_CONFIG_DATA
typedef struct {
  BOOLEAN EnableDnsCache;
  UINT8 Protocol;
  EFI_IPV6_ADDRESS StationIp;
  UINT16 LocalPort;
  UINT32 DnsServerCount;
  EFI_IPV6_ADDRESS *DnsServerList;
  UINT32 RetryCount;
  UINT32 RetryInterval;
} EFI_DNS6_CONFIG_DATA;
```

- **IsDnsServerAuto** If `TRUE`, the DNS server configuration will be retrieved from DHCP server. If `FALSE`, the DNS server configuration will be manually configured through call of `DNSv6.Configure()` interface.
- **EnableDnsCache** If `TRUE`, enable DNS cache function for this DNS instance. If `FALSE`, all DNS query will not lookup local DNS cache.
- **Protocol** Use the protocol number defined in Links to UEFI-Related Documents” (http://uefi.org/uefi) under the heading “IANA Protocol
“Numbers”. Only TCP or UDP are supported, and other protocol values are invalid. An implementation can choose to support only UDP, or both TCP and UDP.

StationIp

The local IP address to use. Set to zero to let the underlying IPv6 driver choose a source address. If not zero it must be one of the configured IP addresses in the underlying IPv6 driver.

DnsServerCount

Count of the DNS servers. When used with GetModeData(), this field is the count of originally configured servers when Configure() was called for this instance. When used with Configure() this is the count of caller-supplied servers. If the DnsServerListCount is zero, the DNS server configuration will be retrieved from DHCP server automatically.

DnsServerList

Pointer to DNS server list containing DnsServerListCount entries or NULL if DnsServerListCount is 0. For Configure(), this will be NULL when there are no caller-supplied server addresses and the DNS instance will retrieve DNS server from DHCP Server. The provided DNS server list is recommended to be filled up in the sequence of preference. When used with GetModeData(), the buffer containing the list will be allocated by the driver implementing this protocol and must be freed by the caller. When used with Configure(), the buffer containing the list will be allocated and released by the caller.

LocalPort

Local port number. Set to zero to use the automatically assigned port number.

RetryCount

Retry number if no response received after RetryInterval.

RetryInterval

Minimum interval of retry is 2 second. If the retry interval is less than 2 second, then use the 2 second.

//**********************************************
// EFI_DNS6_CACHE_ENTRY
//**********************************************
typedef struct {
    CHAR16    *HostName;
    EFI_IPv6_ADDRESS   *IpAddress;
    UINT32    Timeout;
} EFI_DNS6_CACHE_ENTRY;

HostName

Host name. This should be interpreted as Unicode characters.

IpAddress

IP address of this host.

Timeout

Time in second unit that this entry will remain in DNS cache. A value of zero means that this entry is permanent. A nonzero value will override the existing one if this entry to be added is dynamic entry. Implementations may set its default timeout value for the dynamically created DNS cache entry after one DNS resolve succeeds.
Status Codes Returned

<table>
<thead>
<tr>
<th>Code</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>EFI_SUCCESS</td>
<td>The operation completed successfully.</td>
</tr>
<tr>
<td>EFI_NOT_STARTED</td>
<td>When DnsConfigData is queried, no configuration data is available because this instance has not been configured.</td>
</tr>
<tr>
<td>EFI_INVALID_PARAMETER</td>
<td>This is NULL or DnsModeData is NULL.</td>
</tr>
<tr>
<td>EFI_OUT_OF_RESOURCE</td>
<td>Failed to allocate needed resources.</td>
</tr>
</tbody>
</table>

**EFI_DNS6_PROTOCOL.Configure()**

**Summary**

Configure this DNS instance

**Prototype**

```c
typedef EFI_STATUS (EFIAPI * EFI_DNS6_CONFIGURE)(
    IN EFI_DNS6_PROTOCOL      *This,
    IN EFI_DNS6_CONFIG_DATA    *DnsConfigData
);
```

**Descriptions**

The **Configure()** function is used to set and change the configuration data for this EFI DNSv6 Protocol driver instance. Reset the DNS instance if DnsConfigData is NULL.

**Parameters**

- **This**
  
  Pointer to **EFI_DNS6_PROTOCOL** instance.

- **DnsConfigData**
  
  Pointer to the configuration data structure. Type **EFI_DNS6_CONFIG_DATA** is defined in **EFI_DNS6_PROTOCOL.GetModeData()**. All associated storage to be allocated and released by caller.
Status Codes Returned

<table>
<thead>
<tr>
<th>Code</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>EFI_SUCCESS</td>
<td>The operation completed successfully.</td>
</tr>
<tr>
<td>EFI_INVALID_PARAMETER</td>
<td>This is NULL. The StationIp address provided in DnsConfigData is not zero and not a valid unicast. DnsServerList is NULL while DnsServerListCount is not ZERO. DnsServerListCount is zero while DnsServerList is not NULL.</td>
</tr>
<tr>
<td>EFI_OUT_OF_RESOURCES</td>
<td>The DNS instance data or required space could not be allocated.</td>
</tr>
<tr>
<td>EFI_DEVICE_ERROR</td>
<td>An unexpected system or network error occurred. The EFI DNSv6 Protocol instance is not configured.</td>
</tr>
<tr>
<td>EFI_UNSUPPORTED</td>
<td>The designated protocol is not supported.</td>
</tr>
<tr>
<td>EFI_ALREADY_STARTED</td>
<td>Second call to Configure() with DnsConfigData. To reconfigure the instance the caller must call Configure() with NULL first to return driver to unconfigured state.</td>
</tr>
</tbody>
</table>

**EFI_DNS6_PROTOCOL.HostNameToIp()**

**Summary**
Host name to host address translation

**Prototype**
```c
typedef EFI_STATUS
(EFI_API *EFI_DNS6_HOST_NAME_TO_IP) (    
  IN EFI_DNS6_PROTOCOL       *This,
  IN CHAR16            *HostName,
  IN EFI_DNS6_COMPLETION_TOKEN   *Token
);
```

**Parameter**
- **This**
  Pointer to **EFI_DNS6_PROTOCOL** instance.

- **Hostname**
  Pointer to buffer containing fully-qualified Domain Name including Hostname. To resolve successfully, characters within the FQDN string must be chosen according to the format and from within the set of ASCII characters authorized by DNS specifications. Any translation required for reference to domains or hostnames defined as containing Unicode characters, for example use of Punycode, must be performed by caller.

- **Token**
  Point to the completion token to translate host name to host address. Type **EFI_DNS6_COMPLETION_TOKEN** is defined in "Related Definitions" below.
Related Definition

```c
typedef struct {
    EFI_EVENT      Event;
    EFI_STATUS      Status;
    UINT32        RetryCount;
    UINT32        RetryInterval;
    union {
        DNS6_HOST_TO_ADDR_DATA    *H2AData;
        DNS6_ADDR_TO_HOST_DATA    *A2HData;
        DNS6_GENERAL_LOOKUP_DATA   *GLookupData;
    } RspData;
} EFI_DNS6_COMPLETION_TOKEN;
```

**Event**
This `Event` will be signaled after the `Status` field is updated by the EFI DNSv6 protocol driver. The type of `Event` must be `EFI_NOTIFY_SIGNAL`.

**Status**
Will be set to one of the following values.

- **EFI_SUCCESS**: The host name to address translation completed successfully.
- **EFI_NOT_FOUND**: No matching Resource Record (RR) is found.
- **EFI_TIMEOUT**: No DNS server reachable, or `RetryCount` was exhausted without response from all specified DNS servers.
- **EFIDEVICE_ERROR**: An unexpected system or network error occurred.
- **EFI_NO_MEDIA**: There was a media error.

**RetryCount**
The parameter configured through `DNSv6.Configure()` interface. Retry number if no response received after `RetryInterval`.

**RetryInterval**
The parameter configured through `DNSv6.Configure()` interface. Minimum interval of retry is 2 second. If the retry interval is less than 2 second, then use the 2 second.

**H2AData**
When the `Token` is used for host name to address translation, `H2AData` is a pointer to the `DNS6_HOST_TO_ADDR_DATA`. Type `DNS6_HOST_TO_ADDR_DATA` is defined below.

**A2HData**
When the `Token` is used for host address to host name translation, `A2HData` is a pointer to the `DNS6_ADDR_TO_HOST_DATA`. Type `DNS6_ADDR_TO_HOST_DATA` is defined below.

**GLookupDATA**
When the `Token` is used for a general lookup function, `GLookupDATA` is a pointer to the `DNS6_GENERAL_LOOKUP_DATA`. Type `DNS6_GENERAL_LOOKUP_DATA` is defined below.

**EFI_DNS6_COMPLETION_TOKEN** structures are used for host name to address translation, host address to name translation and general lookup operation, the `Event` filed must be filled by the EFI DNSv6 Protocol Client. If the caller attempts to reuse `Token` before the completion event is triggered or
canceled, **EFI_ALREADY_STARTED** will be returned. After the operation completes, the EFI DNSv6 protocol driver fill in the **RspData** and **Status** field and the **Event** is signaled.

```c
//******************************************
// DNS6_HOST_TO_ADDR_DATA
//******************************************
typedef struct {
    UINT32       IpCount;
    EFI_IPv6_ADDRESS   *IpList;
} DNS6_HOST_TO_ADDR_DATA;

IpCount       Number of the returned IP address
IpList       Pointer to the all the returned IP address

//******************************************
// DNS6_ADDR_TO_HOST_DATA
//******************************************
typedef struct {
    CHAR16        *HostName;
} DNS6_ADDR_TO_HOST_DATA;

HostName       Pointer to the primary name for this host address. It’s the caller’s responsibility to free the response memory.

//******************************************
// DNS6_GENERAL_LOOKUP_DATA
//******************************************
typedef struct {
    UINTN          RRCount;
    DNS6_RESOURCE_RECORD  *RRList;
} DNS6_GENERAL_LOOKUP_DATA;

RRCount       Number of returned matching RRs.
RRList       Pointer to the all the returned matching RRs. It’s caller responsibility to free the allocated memory to hold the returned RRs
typedef struct {
    CHAR8 *QName;
    UINT16 QType;
    UINT16 QClass;
    UINT32 TTL;
    UINT16 DataLength;
    CHAR8 *RData;
} DNS6_RESOURCE_RECORD;

QName The Owner name.
QType The Type Code of this RR
QClass The CLASS code of this RR.
TTL 32 bit integer which specify the time interval that the resource record may be cached before the source of the information should again be consulted. Zero means this RR cannot be cached.
DataLength 16 big integer which specify the length of RData.
RData A string of octets that describe the resource, the format of this information varies according to QType and QClass difference.

Description
The HostNameToIp () function is used to translate the host name to host IP address. A type AAAA record query is used to get the one or more IPv6 addresses for this host.

Status Codes Returned

<table>
<thead>
<tr>
<th>Status Code</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>EFI_SUCCESS</td>
<td>The operation was queued successfully.</td>
</tr>
</tbody>
</table>
| EFI_INVALID_PARAMETER | One or more of the following conditions is TRUE
This is NULL.
Token is NULL.
Token. Event is NULL.
HostName is NULL or buffer contained unsupported characters. |
| EFI_NO_MAPPING | There's no source address is available for use. |
| EFI_ALREADY_STARTED | This Token is being used in another DNS session. |
| EFI_NOT_STARTED | This instance has not been started. |
| EFI_OUT_OF_RESOURCES | Failed to allocate needed resources. |
**EFI_DNS6_PROTOCOL.IpToHostName()**

**Summary**

Host address to host name translation

**Prototype**

```c
typedef EFI_STATUS
(EIFIAPI *EFI_DNS6_IP_TO_HOST_NAME) ( 
    IN EFI_DNS6_PROTOCOL       *This,
    IN EFI_IPv6_ADDRESS        IpAddress,
    IN EFI_DNS6_COMPLETION_TOKEN   *Token
);
```

**Parameter**

- **This**: Pointer to EFI_DNS6_PROTOCOL instance.
- **IpAddress**: IP address.
- **Token**: Point to the completion token to translate host address to host name. Type EFI_DNS6_COMPLETION_TOKEN is defined in "Related Definitions" of above HostNameToIp().

**Description**

The IpToHostName() function is used to translate the host address to host name. A type PTR query is used to get the primary name of the host. Implementation can choose to support this function or not.

**Status Codes Returned**

<table>
<thead>
<tr>
<th>Code</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>EFI_SUCCESS</td>
<td>The operation was queued successfully.</td>
</tr>
<tr>
<td>EFI_UNSUPPORTED</td>
<td>This function is not supported</td>
</tr>
<tr>
<td>EFI_INVALID_PARAMETER</td>
<td>One or more of the following conditions is TRUE</td>
</tr>
<tr>
<td></td>
<td>This is NULL.</td>
</tr>
<tr>
<td></td>
<td>Token is NULL.</td>
</tr>
<tr>
<td></td>
<td>Token.Event is NULL.</td>
</tr>
<tr>
<td></td>
<td>IpAddress is not valid IP address.</td>
</tr>
<tr>
<td>EFI_NO_MAPPING</td>
<td>There’s no source address is available for use.</td>
</tr>
<tr>
<td>EFI_NOT_STARTED</td>
<td>This instance has not been started.</td>
</tr>
<tr>
<td>EFI_OUT_OF_RESOURCES</td>
<td>Failed to allocate needed resources.</td>
</tr>
</tbody>
</table>

**EFI_DNS6_PROTOCOL.GeneralLookUp()**

**Summary**

This function provides capability to retrieve arbitrary information from the DNS server.
Prototype

typedef
EFI_STATUS
(EFI_API *EFI_DNS6_GENERAL_LOOKUP) (
    IN EFI_DNS6_PROTOCOL *This,
    IN CHAR8 *QName,
    IN UINT16 QType,
    IN UINT16 QClass,
    IN EFI_DNS6_COMPLETION_TOKEN *Token
);

Description

This **GeneralLookUp()** function retrieves arbitrary information from the DNS. The caller supplies a **QNAME**, **QTYPE**, and **QCLASS**, and all of the matching RRs are returned. All RR content (e.g., TTL) was returned. The caller need parse the returned RR to get required information. The function is optional. Implementation can choose to support it or not.

Parameters

- **This** Pointer to **EFI_DNS6_PROTOCOL** instance.
- **QName** Pointer to Query Name.
- **QType** Query Type.
- **QClass** Query Name.
- **Token** Pointer to the completion token to retrieve arbitrary information. Type **EFI_DNS6_COMPLETION_TOKEN** is defined in "Related Definitions" of above **HostNameToIp()**.

Status Codes Returned

<table>
<thead>
<tr>
<th>Code</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>EFI_SUCCESS</td>
<td>The operation was queued successfully.</td>
</tr>
<tr>
<td>EFI_UNSUPPORTED</td>
<td>This function is not supported. Or the requested QType is not supported.</td>
</tr>
<tr>
<td>EFI_INVALID_PARAMETER</td>
<td>One or more of the following conditions is TRUE:</td>
</tr>
<tr>
<td></td>
<td>This is <strong>NULL</strong>.</td>
</tr>
<tr>
<td></td>
<td>Token is <strong>NULL</strong>.</td>
</tr>
<tr>
<td></td>
<td>Token.Event is <strong>NULL</strong>.</td>
</tr>
<tr>
<td></td>
<td>QName is <strong>NULL</strong>.</td>
</tr>
<tr>
<td>EFI_NO_MAPPING</td>
<td>There’s no source address is available for use.</td>
</tr>
<tr>
<td>EFI_NOT_STARTED</td>
<td>This instance has not been started.</td>
</tr>
<tr>
<td>EFI_OUT_OF_RESOURCES</td>
<td>Failed to allocate needed resources.</td>
</tr>
</tbody>
</table>
EFI_DNS6_PROTOCOL.UpdateDnsCache()

Summary
This function is to update the DNS Cache.

Prototype
typedef
 EFI_STATUS
(EFIAPI *EFI_DNS6_UPDATE_DNS_CACHE) (  
 IN EFI_DNS6_PROTOCOL   *This,  
 IN BOOLEAN DeleteFlag,  
 IN BOOLEAN Override,  
 IN EFI_DNS6_CACHE_ENTRY DnsCacheEntry  
);

Parameters
This Pointer to EFI_DNS6_PROTOCOL instance.
DeleteFlag If FALSE, this function is to add one entry to the DNS Cache. If TRUE, this function will delete matching DNS Cache entry.
Override If TRUE, the matching DNS Cache entry will be overwritten with the supplied parameter. If FALSE, EFI_ACCESS_DENIED will be returned if the entry to be added is already existed.
DnsCacheEntry Pointer to DNS Cache entry.

Description
The UpdateDnsCache() function is used to add/delete/modify DNS cache entry. DNS cache can be normally dynamically updated after the DNS resolve succeeds. This function provided capability to manually add/delete/modify the DNS cache.

Status Codes Returned

<table>
<thead>
<tr>
<th>Code</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>EFI_SUCCESS</td>
<td>The operation completed successfully.</td>
</tr>
</tbody>
</table>
| EFI_INVALID_PARAMETER | One or more of the following conditions is TRUE:  
This is NULL.  
DnsCacheEntry.HostName is NULL.  
DnsCacheEntry.IpAddress is NULL.  
DnsCacheEntry.Timeout is ZERO. |
| EFI_ACCESS_DENIED | The DNS cache entry already exists and Override is not TRUE.                                    |
| EFI_OUT_OF_RESOURCE | Failed to allocate needed resources.                                                            |

EFI_DNS6_PROTOCOL.POLL()

Summary
Polls for incoming data packets and processes outgoing data packets.
Prototype

typedef

EFI_STATUS

(EIFIAPI *EFI_DNS6_POLL) (  
IN EFI_DNS6_PROTOCOL  *This
);

Parameters

This Pointer to EFI_DNS6_PROTOCOL instance.

Description

The Poll() function can be used by network drivers and applications to increase the rate that data packets are moved between the communications device and the transmit and receive queues.

In some systems, the periodic timer event in the managed network driver may not poll the underlying communications device fast enough to transmit and/or receive all data packets without missing incoming packets or dropping outgoing packets. Drivers and applications that are experiencing packet loss should try calling the Poll() function more often.

Status Codes Returned

<table>
<thead>
<tr>
<th>Status Code</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>EFI_SUCCESS</td>
<td>Incoming or outgoing data was processed.</td>
</tr>
<tr>
<td>EFI_NOT_STARTED</td>
<td>This EFI DNS Protocol instance has not been started.</td>
</tr>
<tr>
<td>EFI_INVALID_PARAMETER</td>
<td>This is NULL.</td>
</tr>
<tr>
<td>EFI_NO_MAPPING</td>
<td>There's no source address is available for use.</td>
</tr>
<tr>
<td>EFI_DEVICE_ERROR</td>
<td>An unexpected system or network error occurred.</td>
</tr>
<tr>
<td>EFI_TIMEOUT</td>
<td>Data was dropped out of the transmit and/or receive queue. Consider increasing the polling rate.</td>
</tr>
</tbody>
</table>

EFI_DNS6_PROTOCOL.Cancel()

Abort an asynchronous DNS operation, including translation between IP and Host, and general look up behavior.

EFI Protocol

typedef

EFI_STATUS

(EIFIAPI *EFI_DNS6_CANCEL) (  
IN EFI_DNS6_PROTOCOL  *This,  
IN EFI_DNS6_COMPLETION_TOKEN *Token
);

Parameters

This Pointer to EFI_DNS6_PROTOCOL instance.
Token

Pointer to a token that has been issued by
EFI_DNS6_PROTOCOL.HostNameToIp(),
EFI_DNS6_PROTOCOL.IpToHostName() or
EFI_DNS6_PROTOCOL.GeneralLookUp(). If NULL, all pending
tokens are aborted.

Description
The Cancel() function is used to abort a pending resolution request. After calling this function, Token.Status will be set to EFI_ABORTED and then Token.Event will be signaled. If the token is not in one of the queues, which usually means that the asynchronous operation has completed, this function will not signal the token and EFI_NOT_FOUND is returned.

Status Codes Returned

<table>
<thead>
<tr>
<th>Status Code</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>EFI_SUCCESS</td>
<td>The asynchronous DNS operation was aborted and Token-&gt;Event is signaled.</td>
</tr>
<tr>
<td>EFI_NOT_STARTED</td>
<td>This EFI DNS6 Protocol instance has not been started.</td>
</tr>
<tr>
<td>EFI_INVALID_PARAMETER</td>
<td>This is NULL.</td>
</tr>
<tr>
<td>EFI_NO_MAPPING</td>
<td>There's no source address is available for use.</td>
</tr>
<tr>
<td>EFI_NOT_FOUND</td>
<td>When Token is not NULL and the asynchronous DNS operation was not found in the transmit queue, it is either completed or was not issued by HostNameToIp(), IpToHostName() or GeneralLookUp().</td>
</tr>
</tbody>
</table>

29.6 EFI HTTP Protocols
This section defines the EFI HTTP Protocol interface. It is split into the following two main sections.

- HTTP Service Binding Protocol (HTTPSB)
- HTTP Protocol (HTTP)

29.6.1 HTTP Service Binding Protocol

EFI_HTTP_SERVICE_BINDING_PROTOCOL

Summary
The HTTPSB is used to locate communication devices that are supported by a HTTP driver and to create and destroy instances of the HTTP child protocol driver.

The EFI Service Binding Protocol in Section 2.5.8 defines the generic Service Binding Protocol functions. This section discusses the details that are specific to the HTTP.
GUID
#define EFI_HTTP_SERVICE_BINDING_PROTOCOL_GUID \ 
{0xbdc8e6af, 0xd9bc, 0x4379, \ 
{0xa7, 0x2a, 0xe0, 0xc4, 0xe7, 0x5d, 0xae, 0x1c}}

Description
A network application (or driver) that requires HTTP communication service can use one of the protocol handler services, such as BS->LocateHandleBuffer(), to search for devices that publish a HTTPSB GUID. Each device with a published HTTP SB GUID supports HTTP Service Binding Protocol and may be available for use.

After a successful call to the EFI_HTTP_SERVICE_BINDING_PROTOCOL.CreateChild() function, the child HTTP driver instance is in an uninitialized state; it is not ready to initiate HTTP data transfer.

Before a network application terminates execution, every successful call to the EFI_HTTP_SERVICE_BINDING_PROTOCOL.CreateChild() function must be matched with a call to the EFI_HTTP_SERVICE_BINDING_PROTOCOL.DestroyChild() function.

29.6.2 EFI HTTP Protocol Specific Definitions

EFI_HTTP_PROTOCOL

Protocol GUID
#define EFI_HTTP_PROTOCOL_GUID \ 
{0x7A59B29B, 0x910B, 0x4171, \ 
{0x82, 0x42, 0xA8, 0x5A, 0x0D, 0xF2, 0x5B, 0x5B}}

Protocol Interface Structure
typedef struct _EFI_HTTP_PROTOCOL {
  EFI_HTTP_GET_MODE_DATA GetModeData;
  EFI_HTTP_CONFIGURE Configure;
  EFI_HTTP_REQUEST Request;
  EFI_HTTP_CANCEL Cancel;
  EFI_HTTP_RESPONSE Response;
  EFI_HTTP_POLL Poll;
} EFI_HTTP_PROTOCOL;

Parameters
GetModeData Gets the current operational status. See the GetModeData() function description.
Configure Initialize, change, or reset operational settings in the EFI HTTP protocol instance. See Configure() for function description.
Request Queue a request token into the transmit queue. This function is a non-blocking operation. See Request() for function description.
Cancel Abort a pending request or response operation. See Cancel() for function description.
Response
Queue a response token into the receive queue. This function is a non-blocking operation. See `Response()` for function description.

Poll
Poll to receive incoming HTTP response and transmit outgoing HTTP request. See `Poll()` for function description.

Description
The EFI HTTP protocol is designed to be used by EFI drivers and applications to create and transmit HTTP Requests, as well as handle HTTP responses that are returned by a remote host. This EFI protocol uses and relies on an underlying EFI TCP protocol.

**EFI_HTTP_PROTOCOL.GetModeData()**

Summary
Returns the operational parameters for the current HTTP child instance.

EFI Protocol

```c
typedef
    EFI_STATUS
    (EFIAPI * EFI_HTTP_GET_MODE_DATA)(
        IN EFI_HTTP_PROTOCOL *This,
        OUT EFI_HTTP_CONFIG_DATA *HttpConfigData
    );
```

Parameters
- **This**: Pointer to `EFI_HTTP_PROTOCOL` instance.
- **HttpConfigData**: Pointer to the buffer for operational parameters of this HTTP instance. Type `EFI_HTTP_CONFIG_DATA` is defined in “Related Definitions” below. It is the responsibility of the caller to allocate the memory for `HttpConfigData` and `HttpConfigData->AccessPoint.IPv6Node/IPv4Node`. In fact, it is recommended to allocate sufficient memory to record IPv6Node since it is big enough for all possibilities.

Description
The `GetModeData()` function is used to read the current mode data (operational parameters) for this HTTP protocol instance.

Status Codes Returned

<table>
<thead>
<tr>
<th>Code</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>EFI_SUCCESS</td>
<td>Operation succeeded</td>
</tr>
<tr>
<td>EFI_INVALID_PARAMETER</td>
<td>This is NULL. <code>HttpConfigData</code> is NULL. <code>HttpConfigData-&gt;AccessPoint.IPv4Node</code> or <code>HttpConfigData-&gt;AccessPoint.IPv6Node</code> is NULL</td>
</tr>
<tr>
<td>EFI_NOT_STARTED</td>
<td>This EFI HTTP Protocol instance has not been started.</td>
</tr>
</tbody>
</table>
typedef struct {
    EFI_HTTP_VERSION HttpVersion;
    UINT32 TimeOutMillisec;
    BOOLEAN LocalAddressIsIPv6;
    union {
        EFI_HTTPv4_ACCESS_POINT *IPv4Node;
        EFI_HTTPv6_ACCESS_POINT *IPv6Node;
    }
} EFI_HTTP_CONFIG_DATA;

HttpVersion  HTTP version that this instance will support.
TimeOutMillsec Time out (in milliseconds) when blocking for requests.
LocalAddressIsIPv6 Defines behavior of EFI DNS and TCP protocols consumed by this instance. If FALSE, this instance will use EFI_DNS4_PROTOCOL and EFI_TCP4_PROTOCOL. If TRUE, this instance will use EFI_DNS6_PROTOCOL and EFI_TCP6_PROTOCOL.
IPv4Node When LocalAddressIsIPv6 is FALSE, this points to the local address, subnet, and port used by the underlying TCP protocol.
IPv6Node When LocalAddressIsIPv6 is TRUE, this points to the local IPv6 address and port used by the underlying TCP protocol.

typedef enum {
    HttpVersion10,
    HttpVersion11,
    HttpVersionUnsupported
} EFI_HTTP_VERSION;

typedef struct {
    BOOLEAN UseDefaultAddress;
    EFI_IPv4_ADDRESS LocalAddress;
    EFI_IPv4_ADDRESS LocalSubnet;
    UINT16 LocalPort;
} EFI_HTTPv4_ACCESS_POINT;

UseDefaultAddress Set to TRUE to instruct the EFI HTTP instance to use the default address information in every TCP connection made by this instance.
In addition, when set to TRUE, `LocalAddress` and `LocalSubnet` are ignored.

**LocalAddress**
If `UseDefaultAddress` is set to FALSE, this defines the local IP address to be used in every TCP connection opened by this instance.

**LocalSubnet**
If `UseDefaultAddress` is set to FALSE, this defines the local subnet to be used in every TCP connection opened by this instance.

**LocalPort**
This defines the local port to be used in every TCP connection opened by this instance.

//**********************************************************
// EFI_HTTPv6_ACCESS_POINT
//**********************************************************
typedef struct {
    EFI_IPv6_ADDRESS LocalAddress;
    UINT16 LocalPort;
} EFI_HTTPv6_ACCESS_POINT;

**LocalAddress**
Local IP address to be used in every TCP connection opened by this instance.

**LocalPort**
Local port to be used in every TCP connection opened by this instance.

**EFI_HTTP_PROTOCOL.Configure()**

**Summary**
Initialize or brutally reset the operational parameters for this EFI HTTP instance.

**EFI Protocol**

```
typedef EFI_STATUS (EFIAPI *EFI_HTTP_CONFIGURE)(
    IN EFI_HTTP_PROTOCOL *This,
    IN EFI_HTTP_CONFIG_DATA *HttpConfigData  OPTIONAL
    );
```

**Parameters**

- **This**
  Pointer to **EFI_HTTP_PROTOCOL** instance.
- **HttpConfigData**
  Pointer to the configure data to configure the instance.

**Description**
The `Configure()` function does the following:

- When `HttpConfigData` is not NULL Initialize this EFI HTTP instance by configuring timeout, local address, port, etc.
- When `HttpConfigData` is `NULL`, reset this EFI HTTP instance by closing all active connections with remote hosts, canceling all asynchronous tokens, and flush request and response buffers without informing the appropriate hosts.

No other EFI HTTP function can be executed by this instance until the `Configure()` function is executed and returns successfully.

### Status Codes Returned

<table>
<thead>
<tr>
<th>Code</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>EFI_SUCCESS</code></td>
<td>Operation succeeded.</td>
</tr>
<tr>
<td><code>EFI_INVALID_PARAMETER</code></td>
<td>One or more of the following conditions is <code>TRUE</code>:</td>
</tr>
<tr>
<td></td>
<td>- <code>This</code> is <code>NULL</code>.</td>
</tr>
<tr>
<td></td>
<td>- <code>HttpConfigData-&gt;LocalAddressIsIPv6</code> is <code>FALSE</code> and <code>HttpConfigData-&gt;AccessPoint.IPv4Node</code> is <code>NULL</code>.</td>
</tr>
<tr>
<td></td>
<td>- <code>HttpConfigData-&gt;LocalAddressIsIPv6</code> is <code>TRUE</code> and <code>HttpConfigData-&gt;AccessPoint.IPv6Node</code> is <code>NULL</code>.</td>
</tr>
<tr>
<td><code>EFI_ALREADY_STARTED</code></td>
<td>Reinitialize this HTTP instance without calling <code>Configure()</code> with <code>NULL</code> to reset it.</td>
</tr>
<tr>
<td><code>EFI_DEVICE_ERROR</code></td>
<td>An unexpected system or network error occurred.</td>
</tr>
<tr>
<td><code>EFI_OUT_OF_RESOURCES</code></td>
<td>Could not allocate enough system resources when executing <code>Configure()</code>.</td>
</tr>
<tr>
<td><code>EFI_UNSUPPORTED</code></td>
<td>One or more options in <code>ConfigData</code> are not supported in the implementation.</td>
</tr>
</tbody>
</table>

### `EFI_HTTP_PROTOCOL.Request()`

**Summary**

The `Request()` function queues an HTTP request to this HTTP instance, similar to `Transmit()` function in the EFI TCP driver. When the HTTP request is sent successfully, or if there is an error, `Status` in token will be updated and `Event` will be signaled.

**EFI Protocol**

```c
typedef
  EFI_STATUS
  (EFIAPI *EFI_HTTP_REQUEST) (
    IN EFI_HTTP_PROTOCOL  *This,
    IN EFI_HTTP_TOKEN     *Token
  );
```

**Parameters**

- **This** Pointer to `EFI_HTTP_PROTOCOL` instance.
- **Token** Pointer to storage containing HTTP request token. Type `EFI_HTTP_TOKEN` is defined in "Related Definitions" below.
Related Definition

```c
typedef struct {
  EFI_EVENT Event;
  EFI_STATUS Status;
  EFI_HTTP_MESSAGE *Message;
} EFI_HTTP_TOKEN;
```

**Event**
This `Event` will be signaled after the `Status` field is updated by the EFI HTTP Protocol driver. The type of `Event` must be `EFI_NOTIFY_SIGNAL`. The Task Priority Level (TPL) of `Event` must be lower than or equal to `TPL_CALLBACK`.

**Status**
`Status` will be set to one of the following value if the HTTP request is successfully sent or if an unexpected error occurs:

- **EFI_SUCCESS**: The HTTP request was successfully sent to the remote host.
- **EFI_HTTP_ERROR**: The response message was successfully received but contains a HTTP error. The response status code is returned in `Token`.
- **EFI_ABORTED**: The HTTP request was canceled by the caller and removed from the transmit queue.
- **EFI_TIMEOUT**: The HTTP request timed out before reaching the remote host.
- **EFI_DEVICE_ERROR**: An unexpected system or network error occurred.

**Message**
Pointer to storage containing HTTP message data.

```c
typedef struct {
  union {
    EFI_HTTP_REQUEST_DATA *Request;
    EFI_HTTP_RESPONSE_DATA *Response;
  } Data;
  UINTN HeaderCount;
  EFI_HTTP_HEADER *Headers;
  UINTN BodyLength;
  VOID *Body;
} EFI_HTTP_MESSAGE;
```

**Request**
When the token is used to send a HTTP request, `Request` is a pointer to storage that contains such data as URL and HTTP method.

**Response**
When used to await a response, `Response` points to storage containing HTTP response status code.
HeaderCount  Number of HTTP header structures in Headers list. On request, this count is provided by the caller. On response, this count is provided by the HTTP driver.

Headers  Array containing list of HTTP headers. On request, this array is populated by the caller. On response, this array is allocated and populated by the HTTP driver. It is the responsibility of the caller to free this memory on both request and response.

BodyLength  Length in bytes of the HTTP body. This can be zero depending on the HttpMethod type.

Body  Body associated with the HTTP request or response. This can be NULL depending on the HttpMethod type.

The HTTP driver will prepare a request string from the information contained in and queue it to the underlying TCP instance to be sent to the remote host. Typically, all fields in the structure will contain content (except Body and BodyLength when HTTP method is not POST or PUT), but there is a special case when using PUT or POST to send large amounts of data. Depending on the size of the data, it may not be able to be stored in a contiguous block of memory, so the data will need to be provided in chunks. In this case, if Body is not NULL and BodyLength is non-zero and all other fields are NULL or 0, the HTTP driver will queue the data to be sent to the last remote host that a token was successfully sent. If no previous token was sent successfully, this function will return EFI_INVALID_PARAMETER.

The HTTP driver is expected to close existing (if any) underlying TCP instance and create new TCP instance if the host name in the request URL is different from previous calls to Request(). This is consistent with RFC 2616 recommendation that HTTP clients should attempt to maintain an open TCP connection between client and host.

```c
//**************************************
// EFI_HTTP_REQUEST_DATA
//**************************************
typedef struct {
    EFI_HTTP_METHOD   Method;
    CHAR16       *Url;
} EFI_HTTP_REQUEST_DATA;
```

Method  The HTTP method (e.g. GET, POST) for this HTTP Request.

Url  The URI of a remote host. From the information in this field, the HTTP instance will be able to determine whether to use HTTP or HTTPS and will also be able to determine the port number to use. If no port number is specified, port 80 (HTTP) is assumed. See RFC 3986 for more details on URI syntax.
EFI_HTTP_METHOD

typedef enum {
    HttpMethodGet,
    HttpMethodPost,
    HttpMethodPatch,
    HttpMethodOptions,
    HttpMethodConnect,
    HttpMethodHead,
    HttpMethodPut,
    HttpMethodDelete,
    HttpMethodTrace,
    HttpMethodMax
} EFI_HTTP_METHOD;

EFI_HTTP_RESPONSE_DATA

typedef struct {
    EFI_HTTP_STATUS_CODE StatusCode;
} EFI_HTTP_RESPONSE_DATA;

StatusCode Response status code returned by the remote host.

EFI_HTTP_HEADER

typedef struct {
    CHAR8 *FieldName;
    CHAR8 *FieldValue;
} EFI_HTTP_HEADER;

FieldName Null terminated string which describes a field name. See RFC 2616 Section 14 for detailed information about field names.

FieldValue Null terminated string which describes the corresponding field value. See RFC 2616 Section 14 for detailed information about field values.
typedef enum {
    HTTP_STATUS_UNSUPPORTED_STATUS = 0,
    HTTP_STATUS_100_CONTINUE,
    HTTP_STATUS_101_SWITCHING_PROTOCOLS,
    HTTP_STATUS_200_OK,
    HTTP_STATUS_201_CREATED,
    HTTP_STATUS_202_ACCEPTED,
    HTTP_STATUS_203_NON_AUTHORITATIVE_INFORMATION,
    HTTP_STATUS_204_NO_CONTENT,
    HTTP_STATUS_205_RESET_CONTENT,
    HTTP_STATUS_206_PARTIAL_CONTENT,
    HTTP_STATUS_300_MULTIPLE_CHOICES,
    HTTP_STATUS_301_MOVED_PERMANENTLY,
    HTTP_STATUS_302_FOUND,
    HTTP_STATUS_303_SEE_OTHER,
    HTTP_STATUS_304_NOT_MODIFIED,
    HTTP_STATUS_305_USE_PROXY,
    HTTP_STATUS_307_TEMPORARY_REDIRECT,
    HTTP_STATUS_400_BAD_REQUEST,
    HTTP_STATUS_401_UNAUTHORIZED,
    HTTP_STATUS_402_PAYMENT_REQUIRED,
    HTTP_STATUS_403_FORBIDDEN,
    HTTP_STATUS_404_NOT_FOUND,
    HTTP_STATUS_405_METHOD_NOT_ALLOWED,
    HTTP_STATUS_406_NOT_ACCEPTABLE,
    HTTP_STATUS_407_PROXY_AUTHENTICATION_REQUIRED,
    HTTP_STATUS_408_REQUEST_TIME_OUT,
    HTTP_STATUS_409_CONFLICT,
    HTTP_STATUS_410_GONE,
    HTTP_STATUS_411_LENGTH_REQUIRED,
    HTTP_STATUS_412_PRECONDITION_FAILED,
    HTTP_STATUS_413_REQUEST_ENTITY_TOO_LARGE,
    HTTP_STATUS_414_REQUEST_URI_TOO_LARGE,
    HTTP_STATUS_415_UNSUPPORTED_MEDIA_TYPE,
    HTTP_STATUS_416_REQUESTED_RANGE_NOT_SATISFIED,
    HTTP_STATUS_417_EXPECTATION_FAILED,
    HTTP_STATUS_500_INTERNAL_SERVER_ERROR,
    HTTP_STATUS_501_NOT_IMPLEMENTED,
    HTTP_STATUS_502_BAD_GATEWAY,
    HTTP_STATUS_503_SERVICE_UNAVAILABLE,
    HTTP_STATUS_504_GATEWAY_TIME_OUT,
    HTTP_STATUS_505_HTTP_VERSION_NOT_SUPPORTED,
    HTTP_STATUS_308_PERMANENT_REDIRECT
} EFI_HTTP_STATUS_CODE;
Status Codes Returned

<table>
<thead>
<tr>
<th>Status Code</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>EFI_SUCCESS</td>
<td>Outgoing data was processed.</td>
</tr>
<tr>
<td>EFI_NOT_STARTED</td>
<td>This EFI HTTP Protocol instance has not been started.</td>
</tr>
<tr>
<td>EFI_DEVICE_ERROR</td>
<td>An unexpected system or network error occurred.</td>
</tr>
<tr>
<td>EFI_TIMEOUT</td>
<td>Data was dropped out of the transmit or receive queue.</td>
</tr>
<tr>
<td>EFI_INVALID_PARAMETER</td>
<td>One or more of the following conditions is <strong>TRUE</strong>:</td>
</tr>
<tr>
<td></td>
<td>- <strong>This</strong> is NULL.</td>
</tr>
<tr>
<td></td>
<td>- <strong>Token</strong> is NULL.</td>
</tr>
<tr>
<td></td>
<td>- <strong>Token-&gt;Message</strong> is NULL.</td>
</tr>
<tr>
<td></td>
<td>- <strong>Token-&gt;Message-&gt;Body</strong> is not NULL, <strong>Token-&gt;Message-&gt;BodyLength</strong> is</td>
</tr>
<tr>
<td></td>
<td>non-zero, and <strong>Token-&gt;Message-&gt;Data</strong> is NULL, but a previous call to</td>
</tr>
<tr>
<td></td>
<td><strong>Request()</strong> has not been completed successfully.</td>
</tr>
<tr>
<td>EFI_OUT_OF_RESOURCES</td>
<td>Could not allocate enough system resources.</td>
</tr>
<tr>
<td>EFI_UNSUPPORTED</td>
<td>The HTTP method is not supported in current implementation.</td>
</tr>
</tbody>
</table>

**EFI_HTTP_PROTOCOL.Cancel()**

**Summary**
Abort an asynchronous HTTP request or response token.

**EFI Protocol**
```c
typedef
  EFI_STATUS
  (EFIAPI * EFI_HTTP_CANCEL)(
    IN EFI_HTTP_PROTOCOL  *This,
    IN EFI_HTTP_TOKEN     *Token,
  );
```

**Parameters**
- **This** Pointer to **EFI_HTTP_PROTOCOL** instance.
- **Token** Point to storage containing HTTP request or response token.

**Description**
The **Cancel()** function aborts a pending HTTP request or response transaction. If **Token** is not **NULL** and the token is in transmit or receive queues when it is being cancelled, its **Token->Status** will be set to **EFI_ABORTED** and then **Token->Event** will be signaled. If the token is not in one of the queues, which usually means that the asynchronous operation has completed, **EFI_NOT_FOUND** is returned. If **Token** is **NULL**, all asynchronous tokens issued by **Request()** or **Response()** will be aborted.
Status Codes Returned

<table>
<thead>
<tr>
<th>Code</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>EFI_SUCCESS</td>
<td>Request and Response queues are successfully flushed.</td>
</tr>
<tr>
<td>EFI_INVALID_PARAMETER</td>
<td>This is NULL.</td>
</tr>
<tr>
<td>EFI_NOT_STARTED</td>
<td>This instance hasn’t been configured.</td>
</tr>
<tr>
<td>EFI_NOT_FOUND</td>
<td>The asynchronous request or response token is not found.</td>
</tr>
<tr>
<td>EFI_UNSUPPORTED</td>
<td>The implementation does not support this function.</td>
</tr>
</tbody>
</table>

**EFI_HTTP_PROTOCOL.Response()**

**Summary**

The `Response()` function queues an HTTP response to this HTTP instance, similar to the `Receive()` function in the EFI TCP driver. When the HTTP response is received successfully, or if there is an error, `Status` in token will be updated and `Event` will be signaled.

**EFI Protocol**

```c
typedef EFI_STATUS
    (EFIAPI *EFI_HTTP_RESPONSE) (
    IN EFI_HTTP_PROTOCOL  *This,
    IN EFI_HTTP_TOKEN     *Token
    );
```

**Parameters**

- **This** Pointer to `EFI_HTTP_PROTOCOL` instance.
- **Token** Pointer to storage containing HTTP response token. See `Request()` function for the definition of `EFI_HTTP_TOKEN`.

**Description**

The HTTP driver will queue a receive token to the underlying TCP instance. When data is received in the underlying TCP instance, the data will be parsed and Token will be populated with the response data. If the data received from the remote host contains an incomplete or invalid HTTP header, the HTTP driver will continue waiting (asynchronously) for more data to be sent from the remote host before signaling Event in Token.

It is the responsibility of the caller to allocate a buffer for `Body` and specify the size in `BodyLength`. If the remote host provides a response that contains a content body, up to `BodyLength` bytes will be copied from the receive buffer into `Body` and `BodyLength` will be updated with the amount of bytes received and copied to `Body`. This allows the client to download a large file in chunks instead of into one contiguous block of memory. Similar to HTTP request, if `Body` is not NULL and `BodyLength` is non-zero and all other fields are NULL or 0, the HTTP driver will queue a receive token to underlying TCP instance. If data arrives in the receive buffer, up to `BodyLength` bytes of data will be copied to `Body`. The HTTP driver will then update `BodyLength` with the amount of bytes received and copied to `Body`. 
If the HTTP driver does not have an open underlying TCP connection with the host specified in the response URL, `Response()` will return `EFI_ACCESS_DENIED`. This is consistent with RFC 2616 recommendation that HTTP clients should attempt to maintain an open TCP connection between client and host.

### Status Codes Returned

<table>
<thead>
<tr>
<th>Status Code</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>EFI_SUCCESS</td>
<td>Allocation succeeded</td>
</tr>
<tr>
<td>EFI_NOT_STARTED</td>
<td>This EFI HTTP Protocol instance has not been initialized.</td>
</tr>
<tr>
<td>EFI_INVALID_PARAMETER</td>
<td>One or more of the following conditions is <code>TRUE</code> (This is NULL).</td>
</tr>
<tr>
<td></td>
<td>Token is NULL.</td>
</tr>
<tr>
<td></td>
<td>Token-&gt;Message is NULL.</td>
</tr>
<tr>
<td></td>
<td>Token-&gt;Message-&gt;Body is not NULL.</td>
</tr>
<tr>
<td></td>
<td>Token-&gt;Message-&gt;BodyLength is non-zero, and Token-&gt;Message-&gt;Data is NULL.</td>
</tr>
<tr>
<td></td>
<td>but a previous call to <code>Response()</code> has not been completed successfully.</td>
</tr>
<tr>
<td>EFI_OUT_OF_RESOURCES</td>
<td>Could not allocate enough system resources.</td>
</tr>
<tr>
<td>EFI_ACCESS_DENIED</td>
<td>An open TCP connection is not present with the host specified by response URL.</td>
</tr>
</tbody>
</table>

### EFI_HTTP_PROTOCOL.Poll()

Polls for incoming data packets and processes outgoing data packets.

```c
typedef EFI_STATUS
  (EFIAPI *EFI_HTTP_POLL) (
    IN EFI_HTTP_PROTOCOL*This
  );
```

**Parameters**

- `This` Pointer to `EFI_HTTP_PROTOCOL` instance.

**Description**

The `Poll()` function can be used by network drivers and applications to increase the rate that data packets are moved between the communication devices and the transmit and receive queues. In some systems, the periodic timer event in the managed network driver may not poll the underlying communications device fast enough to transmit and/or receive all data packets without missing incoming packets or dropping outgoing packets. Drivers and applications that are experiencing packet loss should try calling the `Poll()` function more often.
Status Codes Returned

<table>
<thead>
<tr>
<th>Code</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>EFI_SUCCESS</td>
<td>Incoming or outgoing data was processed.</td>
</tr>
<tr>
<td>EFI_INVALID_PARAMETER</td>
<td>This is NULL.</td>
</tr>
<tr>
<td>EFI_DEVICE_ERROR</td>
<td>An unexpected system or network error occurred.</td>
</tr>
<tr>
<td>EFI_NOT_READY</td>
<td>No incoming or outgoing data is processed.</td>
</tr>
<tr>
<td>EFI_NOT_STARTED</td>
<td>This EFI HTTP Protocol instance has not been started.</td>
</tr>
</tbody>
</table>

29.6.2.1 Usage Examples

Here is an example of a client making a HTTP Request to download a driver bundle from Intel Driver Download Center. This example includes sample code for how to support a client that is behind a HTTP proxy server.

```c
#include <Uefi.h>
#include <HttpProtocol.h>

#define BUFFER_SIZE 0x100000

BOOLEAN gRequestCallbackComplete = FALSE;
BOOLEAN gResponseCallbackComplete = FALSE;

VOID
EFIAPI
RequestCallback(
    IN EFI_EVENT Event,
    IN VOID *Context
)
{
    gRequestCallbackComplete = TRUE;
}

VOID
EFIAPI
ResponseCallback(
    IN EFI_EVENT Event,
    IN VOID *Context
)
{
    gResponseCallbackComplete = TRUE;
}

EFI_STATUS
EFI
HttpClientMain(
    IN EFI_HANDLE ImageHandle,
    IN EFI_SYSTEM_TABLE *SystemTable
)
{

```
EFI_STATUS Status;
EFI_SERVICE_BINDING_PROTOCOL *ServiceBinding;
EFI_HANDLE *Handle;
EFI_HTTP_PROTOCOL *HttpProtocol;
EFI_HTTP_CONFIG_DATA ConfigData;
EFI_HTTPv4_ACCESS_POINT IPv4Node;
EFI_HTTP_REQUEST_DATA RequestData;
EFI_HTTP_HEADER RequestHeader;
EFI_HTTP_MESSAGE RequestMessage;
EFI_HTTP_TOKEN RequestToken;
EFI_HTTP_RESPONSE_DATA ResponseData;
EFI_HTTP_MESSAGE ResponseMessage;
EFI_HTTP_TOKEN ResponseToken;
UINT8 *Buffer;
EFI_TIME Baseline;
EFI_TIME Current;
UINTN Timer;
UINTN Index;
UINTN ContentDownloaded;
UINTN ContentLength;

Status = gBS->AllocatePool (EfiBootServicesData,
BUFFER_SIZE,
(VOID **)&Buffer);
// TODO: Handle error...

Status = gBS->LocateProtocol(
&gEfiHttpServiceBindingProtocolGuid,
NULL,
&ServiceBinding);
// TODO: Handle error...

Status = ServiceBinding->CreateChild(ServiceBinding, &Handle);
// TODO: Handle error...

Status = gBS->HandleProtocol(Handle, &gEfiHttpProtocolGuid, &HttpProtocol);
// TODO: Handle error...

ConfigData.HttpVersion = HttpVersion11;
ConfigData.TimeOutMillisec = 0; // Indicates default timeout period
ConfigData.LocalAddressIsIPv6 = FALSE;

ZeroMem(&IPv4Node, sizeof(IPv4Node));
IPv4Node.UseDefaultAddress = TRUE; // Obtain IP address from DHCP
ConfigData.AccessPoint.IPv4Node = &IPv4Node;

// The HTTP driver must first be configured before requests or responses can be processed. This is the same for other network protocols such as TCP.
Status = HttpProtocol->Configure(HttpProtocol, &ConfigData);

// This request message is initialized to request a sample driver bundle
// from Intel's driver download center. To download a file, we use HTTP GET.
RequestData.Method = HttpMethodGet;
// URI where the file is located that we want to download.
// This header tells the HTTP driver to relay the HTTP request
// via a proxy server. This header is just used to demonstrate
// how to relay through a proxy with this driver. The method
// for obtaining the proxy address is up to the client. The
// HTTP driver does NOT resolve this on its own.
RequestHeader.FieldName = "Host";
RequestHeader.FieldValue = "my.proxyserver.com";
// Message format just contains a pointer to the request data
// and body info, if applicable. In the case of HTTP GET, body
// is not relevant.
RequestMessage.Data.Request = &RequestData;
// Just one header being provided in the HTTP message.
RequestMessage.HeaderCount = 1;
RequestMessage.Headers = &RequestHeader;
RequestMessage.BodyLength = 0;
RequestMessage.Body = NULL;
// Token format is similar to the token format in EFI TCP protocol.
RequestToken.Event = NULL;
Status = gBS->CreateEvent(
    EVT_NOTIFY_SIGNAL,
    TPL_CALLBACK,
    RequestCallback,
    NULL,
    &RequestToken.Event
);
// TODO: Handle error...
RequestToken.Status = EFI_SUCCESS;
RequestToken.Message = &RequestMessage;

getRequestCallbackComplete = FALSE;
// Finally, make HTTP request.
Status = HttpProtocol->Request(HttpProtocol, &RequestToken);
// TODO: Handle error...

Status = gRT->GetTime(&Baseline, NULL);
// TODO: Handle error...

// Optionally, wait for a certain amount of time before cancelling
// the request. In this case, we'll allow the network stack 10
// seconds to send the request successfully.
for (Timer = 0; !getRequestCallbackComplete && Timer < 10; ) {
    // Give the HTTP driver some motivation...
HttpProtocol->Poll(HttpProtocol);

// In practice, a call to GetTime() only fails when the total
// elapsed time between the last call to to GetTime() is less
// than the resolution of one tick (e.g. 1 second, depending
// on capabilities of hardware). We only care to check the time
// when the call succeeds.
if (!EFI_ERROR(gRT->GetTime(&Current, NULL)) &&
    Current.Second != Baseline.Second)
{
    // One second has passed, so update Current time and
    // increment the counter.
    Baseline = Current;
    ++Timer;
}

// Cancel request if we did not get a notification from the HTTP
// driver in a timely manner.
if (!gRequestCallbackComplete) {
    Status = HttpProtocol->Cancel(HttpProtocol, &RequestToken);
    // TODO: Handle error and exit condition...
}

// Assuming we succeed in our request...

// This response message is different that request in that the
// HTTP driver is responsible for allocating the headers during
// a response instead of the caller.
ResponseData.StatusCode = HTTP_STATUS_UNSUPPORTED_STATUS;
ResponseMessage.Data.Response = &ResponseData;
// HeaderCount will be updated by the HTTP driver on response.
ResponseMessage.HeaderCount = 0;
// Headers will be populated by the driver on response.
ResponseMessage.Headers = NULL;
// BodyLength maximum limit is defined by the caller. On response,
// the HTTP driver will update BodyLength to the total number of
// bytes copied to Body. This number will never exceed the initial
// maximum provided by the caller.
ResponseMessage.BodyLength = BUFFER_SIZE;
ResponseMessage.Body = Buffer;
// Token format is similar to the token format in EFI TCP protocol.
ResponseToken.Event = NULL;
Status = gBS->CreateEvent(
    EVT_NOTIFY_SIGNAL,
    TPL_CALLBACK,
    NULL,
    &ResponseToken,
    &ResponseToken.Event);
ResponseToken.Status = EFI_SUCCESS;
ResponseToken.Message = &ResponseMessage;
gResponseCallbackComplete = FALSE;
// Finally, make HTTP request.
Status = HttpProtocol->Response(HttpProtocol, &ResponseToken);
// TODO: Handle error...

Status = gRT->GetTime(&Baseline, NULL);
// TODO: Handle error...

// Optionally, wait for a certain amount of time before cancelling.
for (Timer = 0; !gResponseCallbackComplete && Timer < 10; ) {
    HttpProtocol->Poll(HttpProtocol);
    if (!EFI_ERROR(gRT->GetTime(&Current, NULL)) &&
        Current.Second != Baseline.Second)
    {
        // One second has passed, so update Current time and
        // increment the counter.
        Baseline = Current;
        ++Timer;
    }
}

// Remove response token from queue if we did not get a notification
// from the remote host in a timely manner.
if (!gResponseCallbackComplete) {
    Status = HttpProtocol->Cancel(HttpProtocol, &ResponseToken);
    // TODO: Handle error and exit condition...
}

// Assuming we successfully received a response...
for (Index = 0; Index < ResponseMessage.HeaderCount; ++Index) {
    // We can parse the length of the file from the ContentLength header.
    if (!AsciiStriCmp(ResponseMessage.Headers[Index].FieldName, "Content-Length")
    {
        ContentLength =
            AsciiStrDecimalToUintn(ResponseMessage.Headers[Index].FieldValue);
    }
}

ContentDownloaded = ResponseMessage.BodyLength;
// TODO:
// Downloaded data exists in Buffer[0..ResponseMessage.BodyLength].
// At this point, depending on business use case, the content can
// be written to a file, stored on the heap, etc.

while (ContentDownloaded < ContentLength) {
    // If we make it here, we haven't yet downloaded the whole file and
    // need to keep going.
    ResponseMessage.Data.Response = NULL;
    if (ResponseMessage.Headers != NULL) {
        // No sense hanging onto this anymore.
        FreePool(ResponseMessage.Headers);
    }
}
ResponseMessage.HeaderCount = 0;
ResponseMessage.BodyLength = BUFFER_SIZE;
ZeroMem(Buffer, BUFFER_SIZE);
gResponseCallbackComplete = FALSE;
// The HTTP driver accepts a token where Data, Headers, and
// HeaderCount are all 0 or NULL. The driver will wait for a
// response from the last remote host which a transaction occurred
// and copy the response directly into Body, updating BodyLength
// with the total amount copied (downloaded).
Status = HttpProtocol->Response(HttpProtocol, &ResponseToken);
// TODO: Handle error...
Status = gRT->GetTime(&Baseline, NULL);
// TODO: Handle error...

// Optionally, wait for a certain amount of time before cancelling.
for (Timer = 0; !gResponseCallbackComplete && Timer < 10; ) {
    HttpProtocol->Poll(HttpProtocol);
    if (!EFI_ERROR(gRT->GetTime(&Current, NULL)) &&
        Current.Second != Baseline.Second)
    {
        // One second has passed, so update Current time and
        // increment the counter.
        Baseline = Current;
        ++Timer;
    }
}

// Remove response token from queue if we did not get a notification
// from the remote host in a timely manner.
if (!gResponseCallbackComplete) {
    Status = HttpProtocol->Cancel(HttpProtocol, &ResponseToken);
    // TODO: Handle error and exit condition...
}

// Assuming we successfully received a response...
ContentDownloaded += ResponseMessage.BodyLength;
// TODO:
// Downloaded data exists in Buffer[0..ResponseMessage.BodyLength].
// Append data to a file, heap memory, etc.
}

// Perform any necessary cleanup and handling of downloaded file
// assuming we succeeded at downloading the content. Depending on
// where the data was stored as per business need, that data can
// be consumed at this point. For example, if the data was stored
// to a file system, the file can be opened and consumed.
return EFI_SUCCESS;
}

29.6.3 HTTP Utilities Protocol

Summary
This section defines the EFI HTTP Utilities Protocol interface.

**EFI_HTTP_UTILITIES_PROTOCOL**

Protocol GUID

```
#define EFI_HTTP_UTILITIES_PROTOCOL_GUID \
{ 0x3E35C163, 0x4074, 0x45DD,\ 
{ 0x43, 0x1E, 0x23, 0x98, 0x9D, 0xD8, 0x6B, 0x32 }}
```

Protocol Interface Structure

```
typedef struct _EFI_HTTP_UTILITIES_PROTOCOL {
    EFI_HTTP_UTILS_BUILD     Build;
    EFI_HTTP_UTILS_PARSE     Parse;
} EFI_HTTP_UTILITIES_PROTOCOL;
```

Parameters

- **Build**
  Create HTTP header based on a combination of seed header, fields to delete, and fields to append.

- **Parse**
  Parses HTTP header and produces an array of key/value pairs.

Description

The EFI HTTP utility protocol is designed to be used by EFI drivers and applications to parse HTTP headers from a byte stream. This driver is neither dependent on network connectivity, nor the existence of an underlying network infrastructure.

**EFI_HTTP_UTILITIES_PROTOCOL.Build()**

Summary
Provides ability to add, remove, or replace HTTP headers in a raw HTTP message.
EFI Protocol

typedef
EFI_STATUS
(EIFIAPI *EFI_HTTP_UTILS_BUILD) (
    IN EFI_HTTP_UTILITIES_PROTOCOL *This,
    IN UINTN SeedMessageSize
    IN VOID *SeedMessage,  OPTIONAL
    IN UINTN DeleteCount
    IN CHAR8 *DeleteList[],  OPTIONAL
    IN UINTN AppendCount
    IN EFI_HTTP_HEADER [*]AppendList[],  OPTIONAL
    OUT UINTN [*]NewMessageSize,
    OUT VOID **NewMessage,
);  

Parameters

This Pointer to EFI_HTTP_UTILITIES_PROTOCOL instance.
SeedMessageSize Size of the initial HTTP header. This can be zero.
SeedMessage Initial HTTP header to be used as a base for building a new HTTP header. If NULL, SeedMessageSize is ignored.
DeleteCount Number of null-terminated HTTP header field names in DeleteList.
DeleteList List of null-terminated HTTP header field names to remove from SeedMessage. Only the field names are in this list because the field values are irrelevant to this operation.
AppendCount Number of header fields in AppendList.
AppendList List of HTTP headers to populate NewMessage with. If SeedMessage is not NULL, AppendList will be appended to the existing list from SeedMessage in NewMessage
NewMessageSize Pointer to number of header fields in NewMessage.
NewMessage Pointer to a new list of HTTP headers based on

Description

The Build() function is used to manage the headers portion of an HTTP message by providing the ability to add, remove, or replace HTTP headers.
Status Codes Returned

<table>
<thead>
<tr>
<th>Status Code</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>EFI_SUCCESS</td>
<td>Add, remove, and replace operations succeeded.</td>
</tr>
<tr>
<td>EFI_OUT_OF_RESOURCES</td>
<td>Could not allocate memory for <code>NewMessage</code>.</td>
</tr>
<tr>
<td>EFI_INVALID_PARAMETER</td>
<td>One or more of the following conditions is <strong>TRUE</strong>. This is <strong>NULL</strong>.</td>
</tr>
</tbody>
</table>

EFI_HTTP_UTILITIES_PROTOCOL.Parse()

Summary
Parse HTTP header into array of key/value pairs.

EFI Protocol

```c
typedef EFI_STATUS
    (EFIAPI *EFI_HTTP_UTILS_PARSE) ( 
    IN EFI_HTTP_PROTOCOL *This,
    IN CHAR8 *HttpMessage,
    IN UINTN HttpHeadersSize,
    OUT EFI_HTTP_HEADER **HeaderFields,
    OUT UINTN *FieldCount 
    );
```

Parameters

- **This** Pointer to `EFI_HTTP_UTILITIES_PROTOCOL` instance.
- **HttpMessage** Contains raw unformatted HTTP header string.
- **HttpMessageSize** Size of HTTP header.
- **HeaderFields** Array of key/value header pairs.
- **FieldCount** Number of headers in `HeaderFields`.

Description

The `Parse()` function is used to transform data stored in `HttpHeader` into a list of fields paired with their corresponding values.

Status Codes Returned

<table>
<thead>
<tr>
<th>Status Code</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>EFI_SUCCESS</td>
<td>Allocation succeeded</td>
</tr>
<tr>
<td>EFI_NOT_STARTED</td>
<td>This EFI HTTP Protocol instance has not been initialized.</td>
</tr>
<tr>
<td>EFI_INVALID_PARAMETER</td>
<td>One or more of the following conditions is <strong>TRUE</strong>.</td>
</tr>
<tr>
<td></td>
<td>• <strong>This</strong> is <strong>NULL</strong></td>
</tr>
<tr>
<td></td>
<td>• <strong>HttpMessage</strong> is <strong>NULL</strong></td>
</tr>
<tr>
<td></td>
<td>• <strong>HeaderFields</strong> is <strong>NULL</strong></td>
</tr>
<tr>
<td></td>
<td>• <strong>FieldCount</strong> is <strong>NULL</strong></td>
</tr>
</tbody>
</table>
29.7 EFI REST Support Overview

EFI REST(EX) protocols are designed to support REST communication between EFI REST client applications/drivers and REST services. EFI REST client tool uses EFI REST(EX) protocols to send/receive resources to/from REST service to manage systems, configure systems or manipulate resources on REST service. Due to HTTP protocol is commonly used to communicate with REST service in practice, EFI REST(EX) protocols adopt HTTP as the message format to send and receive REST service resource.

EFI REST(EX) driver instance abstracts EFI REST client functionality and provides underlying interface to communicate with REST service. EFI REST(EX) driver instance knows how to communicate with REST service through certain interface after the corresponding configuration is initialized. EFI REST support provides two REST relevant protocols, one is EFI REST protocol which was introduced in UEFI spec 2.5 for providing light-weight EFI REST capability. Another one is EFI REST EX protocol, which is introduced in UEFI spec 2.8 for providing more interoperability between EFI REST client and REST service.

EFI REST and EFI REST EX protocols are not required to coexist on a platform, system integrator determines which EFI REST relevant protocol to be supported on system according to the platform demands. EFI REST support is to provide interoperability between EFI REST client and REST service. The authentication of accessing to REST service is not handled by EFI REST relevant protocols. Different REST service has its own authentication method. EFI REST client has to follow the specification defined by REST service for the authentication process.

Figure 29-1 EFI REST Support, Single Protocol
Multiple EFI REST(EX) driver instances can be installed on a platform to communicate with different types of REST services or various underlying interfaces to REST services. REST service can be located on the platform locally, or off platform in the remote server. The system integrator can implement In-band EFI REST(EX) driver instance for the on-platform REST service communications or Out-of-band EFI REST(EX) driver instance for the off-platform REST service communications.

Figure 29-2 EFI REST Support, Multiple Protocols
**EFI REST Support Scenario 1 (Platform Management)**

The following figure represents a platform which has BMC on board, with the REST service deployed like Redfish service. The In-band EFI REST(EX) protocol (right one) is used by EFI REST client to manage this platform. This platform can also be managed in out of band like from the remote OS REST client. The left one is Out of band EFI REST(EX) protocol which communicate with other REST services like Redfish service in which the resource is belong to other platforms.

![Figure 29-3 EFI REST Support, BMC on Board](image_url)
EFI REST Support Scenario 2 (Platform Management)

The following figure represents a platform which uses remote Redfish service for the platform management. If treats the resource in remote Redfish service as a part of this platform, the In-band EFI REST(EX) protocol could be implemented to communicate with remote Redfish service. This platform can also be managed in out of band from the remote OS REST client.

Figure 29-4 EFI REST Support, Redfish Service
A variety of possible EFI REST(EX) protocol usages are delineated as below. The EFI REST(EX) driver instance could communicate with REST service through underlying interface like EFI network stack, platform specific interface to BMC or others. The working model of EFI REST support depends on the implementation of EFI REST(EX) driver instance and the design of platform.

![Figure 29-5 EFI REST Support, Protocol Usages](image)
29.7.1 EFI REST Protocol

This section defines the EFI REST Protocol interface.

29.7.1.1 EFI REST Protocol Definitions

EFI_REST_PROTOCOL

Protocol GUID

```c
#define EFI_REST_PROTOCOL_GUID
0x0DB48A36, 0x4E54, 0xEA9C,
{ 0x9B, 0x09, 0x1E, 0xA5, 0xBE, 0x3A, 0x66, 0x0B }
```

Protocol Interface Structure

```c
typedef struct _EFI_REST_PROTOCOL {
    EFI_REST_SEND_RECEIVE SendReceive;
    EFI_REST_GET_TIME GetServiceTime;
} EFI_REST_PROTOCOL;
```

Parameters

- **RestSendReceive**: Provides an HTTP-like interface to send and receive resources from a REST service.
- **GetServiceTime**: Returns the current time of the REST service.

Description

The EFI REST protocol is designed to be used by EFI drivers and applications to send and receive resources from a RESTful service. This protocol abstracts REST (Representational State Transfer) client functionality. This EFI protocol could be implemented to use an underlying EFI HTTP protocol, or it could rely on other interfaces that abstract HTTP access to the resources.

**EFI_REST_PROTOCOL.SendReceive()**

Summary

Provides a simple HTTP-like interface to send and receive resources from a REST service.
EFI Protocol

typedef EFI_STATUS (EFIAPI *EFI_REST_SEND_RECEIVE)(
   IN EFI_REST_PROTOCOL *This,
   IN EFI_HTTP_MESSAGE *RequestMessage,
   OUT EFI_HTTP_MESSAGE *ResponseMessage
);

Parameters

This Pointer to EFI_REST_PROTOCOL instance for a particular REST service.
RequestMessage Pointer to the REST request data for this resource
ResponseMessage Pointer to the REST response data obtained for this requested.

Description

The SendReceive() function sends a REST request to this REST service, and returns a REST response when the data is retrieved from the service. Both of the REST request and response messages are represented in format of EFI_HTTP_MESSAGE. RequestMessage contains the request to the REST resource identified by UrlRequestMessage->Data.Request->Url. The ResponseMessage is the returned response for that request, including the final HTTP status code, headers and the REST resource represented in the message body.

The memory buffers pointed by ResponseMessage->Data.Response, ResponseMessage->Headers and ResponseMessage->Body are allocated by this function, and it is the caller's responsibility to free the buffer when the caller no longer requires the buffer's contents.

It's the REST protocol’s responsibility to handle HTTP layer details and return the REST resource to the caller, when this function is implemented by using an underlying EFI HTTP protocol. For example, if an HTTP interim response (Informational 1xx in HTTP 1.1) is received from server, the REST protocol should deal with it and keep waiting for the final response, instead of return the interim response to the caller. Same principle should be observed if the REST protocol relies on other interfaces.

Status Codes Returned

<table>
<thead>
<tr>
<th>Code</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>EFI_SUCCESS</td>
<td>operation succeeded</td>
</tr>
<tr>
<td>EFI_INVALID_PARAMETER</td>
<td>This, RequestMessage, or ResponseMessage are NULL.</td>
</tr>
<tr>
<td>EFI_DEVICE_ERROR</td>
<td>An unexpected system or network error occurred.</td>
</tr>
<tr>
<td>EFI_TIMEOUT</td>
<td>Receiving response message fail due to timeout.</td>
</tr>
</tbody>
</table>
EFI_REST_PROTOCOL.GetServiceTime()

```c
typedef EFI_STATUS (EFIAPI *EFI_REST_GET_TIME)(
    IN   EFI_REST_PROTOCOL   *This,
    OUT  EFI_TIME            *Time
);
```

**Parameters**

- **This**: Pointer to EFI_REST_PROTOCOL instance.
- **Time**: A pointer to storage to receive a snapshot of the current time of the REST service.

**Description**

The `GetServiceTime()` function is an optional interface to obtain the current time from this REST service instance. If this REST service does not support retrieving the time, this function returns EFI_UNSUPPORTED.

**Status Codes Returned**

<table>
<thead>
<tr>
<th>Status Code</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>EFI_SUCCESS</td>
<td>operation succeeded</td>
</tr>
<tr>
<td>EFI_INVALID_PARAMETER</td>
<td>This or Time are NULL</td>
</tr>
<tr>
<td>EFI_UNSUPPORTED</td>
<td>The RESTful service does not support returning the time</td>
</tr>
<tr>
<td>EFI_DEVICE_ERROR</td>
<td>An unexpected system or network error occurred.</td>
</tr>
</tbody>
</table>

### 29.7.2 EFI REST EX Protocol

This section defines the EFI REST EX Protocol interfaces. It is split into the following two main sections:

- REST EX Service Binding Protocol (RESTEXSB)
- REST EX Protocol (REST EX)

#### 29.7.2.1 REST EX Service Binding Protocol

**EFI_REST_EX_SERVICE_BINDING_PROTOCOL**

**Summary**

The RESTEXSB is used to locate the REST services those are supported by a REST EX driver instances and to create and destroy instances of REST EX child protocol driver.

The EFI Service Binding Protocol in Section 11.6 defines the generic Service Binding Protocol functions. This section discusses the details that are specific to the REST EX.
**GUID**

```c
#define EFI_REST_EX_SERVICE_BINDING_PROTOCOL_GUID  
   {0x456bbe01, 0x99d0, 0x45ea,  
   {0xbb, 0x5f, 0x16, 0xd8, 0x4b, 0xed, 0xc5, 0x59}}
```

**Description**

A REST service client application (or driver) that communicates to REST service can use one of protocol handler services, such as BS->LocateHandleBuffer(), to search for devices that publish a RESTEXSB GUID. Each device with a published RESTEXSB GUID supports REST EX Service Binding Protocol and may be available for use.

After a successful call to the EFI_REST_EX_SERVICE_BINDING_PROTOCOL.CreateChild() function, the child REST EX driver is in the unconfigured state. It is not ready to communicate with REST service at this moment. The child instance is ready to use to communicate with REST service after the successful Configure() is invoked. For EFI REST drivers which don’t require additional configuration process, Configure() is unnecessary to be invoked before using its child instance. This depends on EFI REST EX driver specific implementation.

Before a REST service client application terminates execution, every successful call to the EFI_REST_EX_SERVICE_BINDING_PROTOCOL.CreateChild() function must be matched with a call to the EFI_REST_EX_SERVICE_BINDING_PROTOCOL.DestroyChild() function.

### 29.7.2.2 REST EX Protocol Specific Definitions

**EFI_REST_EX_PROTOCOL**

**Protocol GUID**

```c
#define EFI_REST_EX_PROTOCOL_GUID  
   {0x55648b91, 0xe7d, 0x40a3,  
   {0xa9, 0xb3, 0xa8, 0x15, 0xd7, 0xea, 0xdf, 0x97}}
```

**Protocol Interface Structure**

```c
typedef struct _EFI_REST_EX_PROTOCOL {
   EFI_REST_SEND_RECEIVE SendReceive;
   EFI_REST_GET_TIME GetServiceTime;
   EFI_REST_EX_GET_SERVICE GetService;
   EFI_REST_EX_GET_MODE_DATA GetDataMode;
   EFI_REST_EX_CONFIGURE Configure;
   EFI_REST_EX_ASYNC_SEND_RECEIVE AsyncSendReceive;
   EFI_REST_EX_EVENT_SERVICE EventService;
} EFI_REST_EX_PROTOCOL;
```

**Parameters**

- **SendReceive** Provides an HTTP-like interface to send and receive resources from a REST service. The functionality of this function is same as EFI_REST_PROTOCOL.SendReceive(). Refer to section Section 29.7.1.1 for more details.
**GetServiceTime**

Returns the current time of the REST service. The functionality of this function is same as EFI_REST_PROTOCOL.GetServiceTime(). Refer to Section 29.7.1.1 for the details.

**GetService**

This function returns the type and location of REST service.

**GetModeData**

This function returns operational configuration of current EFI REST EX child instance.

**Configure**

This function is used to configure EFI REST EX child instance.

**AsyncSendReceive**

Provides an HTTP-like interface to send and receive resources. The resource returned from REST service is sent to client in asynchronously.

**EventService**

Provides an interface to subscribe event of specific resource changes on REST service.

**Description**

The REST EX protocol is designed to use by REST service client applications or drivers to communicate with REST service. REST EX protocol enhances the REST protocol and provides comprehensive interfaces to REST service clients. Akin to REST protocol, REST EX driver instance uses HTTP message for the REST request and response. However, the underlying mechanism of REST EX is not necessary to be HTTP-aware. The underlying mechanism could be any protocols according to the REST service mechanism respectively. REST EX protocol could be used to communicate with In-band or Out-of-band REST service depends on the platform-specific implementation.

**EFI_REST_EX_PROTOCOL.SendReceive()**

**Summary**

Provides a simple HTTP-like interface to send and receive resources from a REST service.

**Prototype**

```c
typedef
EFI_STATUS
(EFIAPI *EFI_REST_SEND_RECEIVE)(
    IN     EFI_REST_EX_PROTOCOL    *This,
    IN     EFI_HTTP_MESSAGE        *RequestMessage,
    OUT    EFI_HTTP_MESSAGE        *ResponseMessage
);
```

**Parameters**

Refer to Section 29.7.1.1 for the details.

**Description**

Refer to Section 29.7.1.1 for the details.
Status Codes Returned

<table>
<thead>
<tr>
<th>Code</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>EFI_SUCCESS</td>
<td>Operation succeeded</td>
</tr>
<tr>
<td>EFI_INVALID_PARAMETER</td>
<td>This, RequestMessage, or ResponseMessage are NULL.</td>
</tr>
<tr>
<td>EFI_DEVICE_ERROR</td>
<td>An unexpected system or network error occurred.</td>
</tr>
<tr>
<td>EFI_NOT_READY</td>
<td>The configuration of this instance is not set yet. Configure() must be executed and returns successfully prior to invoke this function.</td>
</tr>
<tr>
<td>EFI_TIMEOUT</td>
<td>Receiving response message fail due to timeout.</td>
</tr>
</tbody>
</table>

EFI_REST_EX_PROTOCOL.GetService()

Summary
This function returns the information of REST service provided by this EFI REST EX driver instance.

Protocol Interface
typedef
  EFI_STATUS
  (EFIAPI *EFI_REST_EX_GET_SERVICE)(
    IN EFI_REST_EX_PROTOCOL *This,
    OUT EFI_REST_EX_SERVICE_INFO **RestExServiceInfo
  );

Parameters
This
This is the EFI_REST_EX_PROTOCOL instance.

RestExServiceInfo
Pointer to receive a pointer to EFI_REST_SERVICE_INFO structure. The format of EFI_REST_SERVICE_INFO is version controlled for the future extension. The version of EFI_REST_SERVICE_INFO structure is returned in the header within this structure. EFI REST client refers to the correct format of structure according to the version number. The pointer to EFI_REST_SERVICE_INFO is a memory block allocated by EFI REST EX driver instance. That is caller’s responsibility to free this memory when this structure is no longer needed. Refer to Related Definitions below for the definitions of EFI_REST_SERVICE_INFO structure.

Description
This function returns the information of REST service provided by this REST EX driver instance. The information such as the type of REST service and the access mode of REST EX driver instance (In-band or Out-of-band) are described in EFI_REST_SERVICE_INFO structure. For the vendor-specific REST service, vendor-specific REST service information is returned in VendorSpecificData. Besides the REST service information provided by REST EX driver instance, EFI_DEVICE_PATH_PROTOCOL of the REST service is also provided on the handle of REST EX driver instance.

EFI REST client can get the information of REST service from REST service EFI device path node in EFI_DEVICE_PATH_PROTOCOL. EFI_DEVICE_PATH_PROTOCOL which installed on REST EX driver
instance indicates where the REST service is located, such as BMC Device Path, IPV4, IPV6 or others. Refer to Section 10.3.4.32 for details of the REST service device path node, which is the sub-type (Sub-type = 32) of Messaging Device Path (type 3).

REST EX driver designer is well know what REST service this REST EX driver instance intends to communicate with. The designer also well know this driver instance is used to talk to BMC through specific platform mechanism or talk to REST server through UEFI HTTP protocol. REST EX driver is responsible to fill up the correct information in EFI_REST_EX_SERVICE_INFO. EFI_REST_EX_SERVICE_INFO is referred by EFI REST clients to pickup the proper EFI REST EX driver instance to get and set resource. GetService() is a basic and mandatory function which must be able to use even Configure() is not invoked in previously.

Related Definitions

```c
//*******************************************************
//EFI_REST_EX_SERVICE_INFO_HEADER
//*******************************************************
typedef struct {
    UINT32         Length;
    EFI_REST_EX_SERVICE_INFO_VER   RestServiceInfoVer;
} EFI_REST_EX_SERVICE_INFO_HEADER;

Length          The length of entire EFI REST_EX_SERVICE_INFO structure. Header size is included.
RestServiceInfoVer  The version of this EFI_REST_SERVICE_INFO structure. See below definitions of EFI_REST_EX_SERVICE_INFO_VER.

//*******************************************************
//EFI_REST_EX_SERVICE_INFO_VER
//*******************************************************
typedef struct {
    UINT8         Major;
    UINT8         Minor;
} EFI_REST_EX_SERVICE_INFO_VER;

Major          The major version of EFI_REST_EX_SERVICE_INFO.
Minor          The minor version of EFI_REST_EX_SERVICE_INFO.

//*******************************************************
//EFI_REST_EX_SERVICE_INFO
//*******************************************************
EFI_REST_EX_SERVICE_INFO is version controlled for the future extensions. Any new information added to this structure requires version increased. EFI REST EX driver instance must provides the correct version of structure in EFI_REST_EX_SERVICE_INFO_VER when it returns EFI_REST_EX_SERVICE_INFO to caller.
```
typedef union {
    EFI_REST_EX_SERVICE_INFO_HEADER EfiRestExServiceInfoHeader;
    EFI_REST_EX_SERVICE_INFO_V_1_0 EfiRestExServiceInfoV10;
} EFI_REST_EX_SERVICE_INFO;

typedef struct {
    EFI_REST_EX_SERVICE_INFO_HEADER EfiRestExServiceInfoHeader;
    EFI_REST_EX_SERVICE_TYPE RestExServiceType;
    EFI_REST_EX_SERVICE_ACCESS_MODE RestServiceAccessMode;
    EFI_GUID VendorRestServiceName;
    UINT32 VendorSpecificDataLength;
    UINT8 *VendorSpecificData;
    EFI_REST_EX_CONFIG_TYPE RestExConfigType;
    UINT8 RestExConfigDataLength;
} EFI_REST_EX_SERVICE_INFO_V_1_0;

EfiRestExServiceInfoHeader The header of EFI_REST_EX_SERVICE_INFO.
RestExServiceType The REST service type. See below definition.
RestServiceAccessMode The access mode of REST service. See below definition.
VendorRestServiceName The name of vendor-specific REST service. This field is only valid if RestExServiceType is EFI_REST_EX_SERVICE_VENDOR_SPECIFIC.
VendorSpecificDataLength The length of vendor-specific REST service information. This field is only valid if RestExServiceType is EFI_REST_EX_SERVICE_VENDOR_SPECIFIC.
VendorSpecificData A pointer to vendor-specific REST service information. This field is only valid if RestExServiceType is EFI_REST_EX_SERVICE_VENDOR_SPECIFIC. The memory buffer pointed by VendorSpecificData is allocated by EFI REST EX driver instance and must be freed by EFI REST client when it is no longer need.
RestExConfigType The type of configuration of REST EX driver instance. See GetModeData() and Configure() for the details.
RestExConfigDataLength The length of REST EX configuration data.
typedef enum {
  EFI_REST_EX_SERVICE_UNSPECIFIC = 1,
  EFI_REST_EX_SERVICE_REDFISH,
  EFI_REST_EX_SERVICE_ODATA,
  EFI_REST_EX_SERVICE_VENDOR_SPECIFIC = 0xff,
  EFI_REST_EX_SERVICE_TYPE_MAX
} EFI_REST_EX_SERVICE_TYPE;

EFI_REST_EX_SERVICE_UNSPECIFIC indicates this EFI REST EX driver instance is not used to communicate with any particular REST service. The EFI REST EX driver instance which reports this service type is REST service independent and only provides SendReceive() function to EFI REST client. EFI REST client uses this function to send and receive HTTP message to any target URI and handles the follow up actions by itself. The EFI REST EX driver instance in this type must returns EFI_UNSUPPORTED in below REST EX protocol interfaces, GetServiceTime(), AsyncSendReceive() and EventService().

EFI_REST_EX_SERVICE_REDFISH indicates this EFI REST EX driver instance is used to communicate with Redfish REST service.

EFI_REST_EX_SERVICE_ODATA indicates this EFI REST EX driver instance is used to communicate with Odata REST service.

EFI_REST_EX_SERVICE_VENDOR_SPECIFIC indicates this EFI REST EX driver instance is used to communicate with vendor-specific REST service.

typedef enum {
  EFI_REST_EX_SERVICE_IN_BAND_ACCESS = 1,
  EFI_REST_EX_SERVICE_OUT_OF_BAND_ACCESS = 2,
  EFI_REST_EX_SERVICE_ACCESS_MODE_MAX
} EFI_REST_EX_SERVICE_ACCESS_MODE;

EFI_REST_EX_SERVICE_IN_BAND_ACCESS mode indicates the REST service is invoked in In-band mechanism in the scope of platform. In most of cases, the In-band mechanism is used to communicate with REST service on platform through some particular devices like BMC, Embedded Controller and other infrastructures built on the platform.

EFI_REST_EX_SERVICE_OUT_OF_BAND_ACCESS mode indicates the REST service is invoked in Out-of-band mechanism. The REST service is located out of platform scope. In most of cases, the Out-of-band mechanism is used to communicate with REST service on other platforms through network or other protocols.
EDI REST EX CONFIG TYPE

typedef enum {
    EFI_REST_EX_CONFIG_TYPE_HTTP,
    EFI_REST_EX_CONFIG_TYPE_UNSPECIFIC,
    EFI_REST_EX_CONFIG_TYPE_MAX
} EFI_REST_EX_CONFIG_TYPE;

EFI_REST_EX_CONFIG_TYPE_HTTP indicates the format of the REST EX configuration is
EFI_REST_EX_HTTP_CONFIG_DATA. RestExConfigDataLength of this type is the size of
EFI_REST_EX_HTTP_CONFIG_DATA. This configuration type is used for the HTTP-aware EFI REST EX
driver instance.

EDI REST EX HTTP_CONFIG_DATA

typedef struct {
    EFI_HTTP_CONFIG_DATA HttpConfigData;
    UINT32 SendReceiveTimeout;
} EFI_REST_EX_HTTP_CONFIG_DATA;

HttpConfigData Parameters to configure the HTTP child instance.
SendReceiveTimeout Time out (in milliseconds) when blocking for response after send out
request message in EFI_REST_EX_PROTOCOL.SendReceive().

EDI REST EX_CONFIG_TYPE_UNSPECIFIC indicates the format of REST EX configuration is unspecific.
RestExConfigDataLength of this type depends on the implementation of non HTTP-aware EFI REST EX
driver instance such as BMC EFI REST EX driver instance. The format of configuration for this type refers
to the system/platform spec which is out of UEFI scope.

Status Code Returned

<table>
<thead>
<tr>
<th>Status Code</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>EFI_SUCCESS</td>
<td>EFI_REST_EX_SERVICE_INFO is returned in RestExServiceInfo.</td>
</tr>
<tr>
<td>EFI_UNSUPPORTED</td>
<td>This function is not supported in this REST EX Protocol driver instance.</td>
</tr>
</tbody>
</table>

EFI_REST_EX_PROTOCOL.GetModeData()

Summary
This function returns operational configuration of current EFI REST EX child instance.
Protocol Interface

```c
typedef EFI_STATUS (EFIAPI *EFI_REST_EX_GET_MODE_DATA)(
    IN   EFI_REST_EX_PROTOCOL  *This,
    OUT  EFI_REST_EX_CONFIG_DATA *RestExConfigData
);
```

Parameters

- `This` This is the `EFI_REST_EX_PROTOCOL` instance.
- `RestExConfigData` Pointer to receive a pointer to `EFI_REST_EX_CONFIG_DATA`. The memory allocated for configuration data should be freed by caller. See Related Definitions for the details.

Description

This function returns the current configuration of EFI REST EX child instance. The format of operational configuration depends on the implementation of EFI REST EX driver instance. For example, HTTP-aware EFI REST EX driver instance uses EFI HTTP protocol as the underlying protocol to communicate with the REST service. In this case, the type of configuration `EFI_REST_EX_CONFIG_TYPE_HTTP` is returned from `GetService()`. `EFI_REST_EX_HTTP_CONFIG_DATA` is used as EFI REST EX configuration format and returned to the EFI REST client. For those non HTTP-aware REST EX driver instances, the type of configuration `EFI_REST_EX_CONFIG_TYPE_UNSPECIFIC` is returned from `GetService()`. In this case, the format of returning data could be non-standard. Instead, the format of configuration data is a system/platform specific definition such as a BMC mechanism used in EFI REST EX driver instance. EFI REST client and EFI REST EX driver instance have to refer to the specific system/platform spec which is out of UEFI scope.

Related Definitions

```c
//******************************************************
//EFI_REST_EX_CONFIG_DATA
//******************************************************
typedef UINT8 *EFI_REST_EX_CONFIG_DATA;
```

Status Code Returned

<table>
<thead>
<tr>
<th>Status Code</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>EFI_SUCCESS</td>
<td><code>EFI_REST_EX_SERVICE_INFO</code> is returned in <code>RestExServiceInfo</code>.</td>
</tr>
<tr>
<td>EFI_UNSUPPORTED</td>
<td>This function is not supported in this REST EX Protocol driver instance.</td>
</tr>
<tr>
<td>EFI_NOT_READY</td>
<td>The configuration of this instance is not set yet. <code>Configure()</code> must be executed and return successfully prior to invoke this function</td>
</tr>
</tbody>
</table>

**EFI_REST_EX_PROTOCOL.Configure()**

Summary

This function is used to configure EFI REST EX child instance.
Protocol Interface

```c
typedef EFI_STATUS (EFIAPI *EFI_REST_EX_CONFIGURE)(
    IN EFI_REST_EX_PROTOCOL *This,
    IN EFI_REST_EX_CONFIG_DATA RestExConfigData
);
```

**Parameters**

- **This**
  - This is the `EFI_REST_EX_PROTOCOL` instance.

- **RestExConfigData**
  - Pointer to `EFI_REST_EX_CONFIG_DATA`. See Related Definitions in `GetModeData()` protocol interface.

**Description**

This function is used to configure the setting of underlying protocol of REST EX child instance. The type of configuration is according to the implementation of EFI REST EX driver instance. For example, HTTP-aware EFI REST EX driver instance uses EFI HTTP protocol as the undying protocol to communicate with REST service. The type of configuration is `EFI_REST_EX_CONFIG_TYPE_HTTP` and `RestExConfigData` is in the format of `EFI_REST_EX_HTTP_CONFIG_DATA`.

Akin to HTTP configuration, REST EX child instance can be configure to use different HTTP local access point for the data transmission. Multiple REST clients may use different configuration of HTTP to distinguish themselves, such as to use the different TCP port. For those non HTTP-aware REST EX driver instance, the type of configuration is `EFI_REST_EX_CONFIG_TYPE_UNSPECIFIC`. `RestExConfigData` refers to the non industrial standard. Instead, the format of configuration data is system/platform specific definition such as BMC. In this case, EFI REST client and EFI REST EX driver instance have to refer to the specific system/platform spec which is out of the UEFI scope. Besides `GetService()` function, no other EFI REST EX functions can be executed by this instance until `Configure()` is executed and returns successfully. All other functions must returns `EFI_NOT_READY` if this instance is not configured yet. Set `RestExConfigData` to NULL means to put EFI REST EX child instance into the unconfigured state.

**Status Code Returned**

<table>
<thead>
<tr>
<th>Status Code</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>EFI_SUCCESS</td>
<td><code>EFI_REST_EX_CONFIG_DATA</code> is set in successfully.</td>
</tr>
<tr>
<td>EFI_DEVICE_ERROR</td>
<td>Configuration for this REST EX child instance is failed with the given <code>EFI_REST_EX_CONFIG_DATA</code>.</td>
</tr>
<tr>
<td>EFI_UNSUPPORTED</td>
<td>This function is not supported in this REST EX Protocol driver instance.</td>
</tr>
</tbody>
</table>
**Usage Example**

Below illustrations show the usage cases of using different EFI REST EX child instances to communicate with REST service.

In the above case, EFI REST Client A and B use HTTP-aware EFI REST EX driver instance to get and send resource. These two EFI REST clients configure the child instance with specific TCP port. Therefore the data transmission through HTTP can delivered to the proper EFI REST clients.

In the above case, EFI REST Client A creates two EFI REST EX child instances and configures those child instances to connect to two BMCs respectively.

**EFI_REST_EX_PROTOCOL.AsyncSendReceive()**

**Summary**

This function sends REST request to REST service and signal caller’s event asynchronously when the final response is received by REST EX Protocol driver instance. The essential design of this function is to handle asynchronous send/receive implicitly according to REST service asynchronous request mechanism. Caller will get the notification once the final response is returned from the REST service.
Protocol Interface

```c
typedef EFI_STATUS (EFIAPIC *EFI_REST_EX_ASYNC_SEND_RECEIVE)(
    IN EFI_REST_EX_PROTOCOL *This,
    IN EFI_HTTP_MESSAGE *RequestMessage OPTIONAL,
    IN EFI_REST_EX_TOKEN *RestExToken,
    IN UINTN *TimeOutInMilliSeconds OPTIONAL);
```

Parameters

- `This` is the `EFI_REST_EX_PROTOCOL` instance.
- `RequestMessage` is the REST request message sent to the REST service. Set `RequestMessage` to NULL to cancel the previous asynchronous request associated with the corresponding `RestExToken`. See descriptions for the details.
- `RestExToken` is the REST EX token which REST EX Protocol instance uses to notify REST client the status of response of asynchronous REST request. See related definition of `EFI_REST_EX_TOKEN`.
- `TimeOutInMilliSeconds` is the pointer to the timeout in milliseconds which REST EX Protocol driver instance refers as the duration to drop asynchronous REST request. NULL pointer means no timeout for this REST request. REST EX Protocol driver signals caller's event with `EFI_STATUS` set to `EFI_TIMEOUT` in `RestExToken` if REST EX Protocol can't get the response from REST service within `TimeOutInMilliSeconds`.

Description

This function is used to send REST request with asynchronous REST service response within certain timeout declared. REST service sometime takes long time to create resource. Sometimes REST service returns response to REST client late because of the shortage of bandwidth or bad network quality. To prevent from unfriendly user experience due to system stuck while waiting for the response from REST service, `EFI_REST_EX_PROTOCOL.AsyncSendReceive()` provides the capability to send asynchronous REST request. Caller sends the REST request and still can execute some other processes on background while waiting the event signaled by REST EX Protocol driver instance.

The implementation of underlying mechanism of asynchronous REST request depends on the mechanism of REST service. HTTP protocol, In-Band management protocol and other protocols has its own way to support asynchronous REST request. Similar to `EFI_REST_EX_PROTOCOL.SendReceive()`, it’s the REST EX protocol’s responsibility to handle the implementation details and return only the REST resource to the caller. REST EX Protocol driver instance which doesn’t support asynchronous REST request can just return `EFI_UNSUPPORTED` to caller. Also, this function must returns `EFI_UNSUPPORTED` if `EFI_REST_EX_SERVICE_TYPE` returned in `EFI_REST_EX_SERVICE_INFO` from `GetService()` is `EFI_REST_EX_SERVICE_UNSPECIFIC`.

REST clients do not have to know the preprocessors of asynchronous REST request between REST EX Protocol driver instance and REST service. The responsibility of REST EX Protocol driver instance is to monitor the status of resource readiness and to signal caller’s `RestExToken` when the status of returning
resource is ready. REST EX Protocol driver instance sets Status field in RestExToken to EFI_SUCCESS and sets ResponseMessage pointer to the final response from REST service. Then signal caller’s event to notify REST client the desired REST resource is received. REST EX Protocol driver instance also has to create an EFI timer to handle the timeout situation. REST EX Protocol driver must drops the asynchronous REST request once the timeout is expired. In this case, REST EX Protocol driver instance sets Status field in RestExToken to EFI_TIMEOUT and signal caller’s event token.

REST EX Protocol driver instance must has capability to cancel the in process asynchronous REST request when caller asks to terminate specific asynchronous REST request. REST EX Protocol driver instance may not have capability to force REST service to cancel the specific request, however, REST EX Protocol driver instance at least least can clean up its own internal resource of asynchronous REST request. Caller has to set RequestMessage to NULL with RestExToken set to EFI_REST_EX_TOKEN which was successfully sent to this function previously. REST EX Protocol driver instance finds the given EFI_REST_EX_TOKEN from its private database and clean up the associated resource if EFI_REST_EX_TOKEN is an in-process asynchronous REST request. REST EX Protocol driver instance then sets Status field in RestExToken to EFI_ABORT and signal caller’s event to indicate the asynchronous REST request has been canceled.

REST EX Protocol driver instance maintains the internal property, state machine, status of transfer of each asynchronous REST request. REST EX Protocol driver instance has to clean up the internal resource associated with each asynchronous REST request no matter the transfer is ended with success or fail.

There are two phases of asynchronous REST request. One is the preprocessor of establishing asynchronous REST request between REST EX Protocol driver instance and REST service. Another phase is to retrieve the final response from REST service and send to REST client.

**Related Definitions**

```c
typedef struct {
  EFI_EVENT Event;
  EFI_STATUS Status;
  EFI_HTTP_MESSAGE *ResponseMessage;
} EFI_REST_EX_TOKEN;
```

This event will be signaled after the Status field is updated by the EFI REST EX Protocol driver instance. The type of Event must be EFI_NOTIFY_SIGNAL. The Task Priority Level (TPL) of Event must be lower than or equal to TPL_CALLBACK, which allows other events to be notified.
**Status**

Status will be set to one of the following values if the REST EX Protocol driver instance gets the response from the REST service successfully, or if an unexpected error occurs:

**EFI_SUCCESS**: The resource gets a response from REST service successfully. **ResponseMessage** points to the response in HTTP message structure.

**EFI_ABORTED**: The asynchronous REST request was canceled by the caller.

**EFI_TIMEOUT**: The asynchronous REST request timed out before receiving a response from the REST service.

**EFI_DEVICE_ERROR**: An unexpected error occurred.

**ResponseMessage**

The REST response message pointed to by this pointer is only valid when **Status** is **EFI_SUCCESS**. The memory buffers pointed to by **ResponseMessage**, **ResponseMessage->Data.Response**, **ResponseMessage->Headers** and **ResponseMessage->Body** are allocated by the EFI REST EX driver instance, and it is the caller's responsibility to free the buffer when the caller no longer requires the buffer's contents.

### Status Code Returned

<table>
<thead>
<tr>
<th>Status Code</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>EFI_SUCCESS</td>
<td>Asynchronous REST request is established.</td>
</tr>
<tr>
<td>EFI_UNSUPPORTED</td>
<td>This REST EX Protocol driver instance doesn’t support asynchronous request.</td>
</tr>
<tr>
<td>EFI_TIMEOUT</td>
<td>Asynchronous REST request is not established and timeout is expired.</td>
</tr>
<tr>
<td>EFI_ABORT</td>
<td>Previous asynchronous REST request has been canceled.</td>
</tr>
<tr>
<td>EFI_DEVICE_ERROR</td>
<td>Otherwise, returns <strong>EFI_DEVICE_ERROR</strong> for other errors according to HTTP Status Code.</td>
</tr>
<tr>
<td>EFI_NOT_READY</td>
<td>The configuration of this instance is not set yet. <strong>Configure()</strong> must be executed and returns successfully prior to invoke this function.</td>
</tr>
</tbody>
</table>

### EFI_REST_EX_PROTOCOL.EventService()

**Summary**

This function sends REST request to a REST Event service and signals caller’s event token asynchronously when the URI resource change event is received by REST EX Protocol driver instance. The essential design of this function is to monitor event implicitly according to REST service event service mechanism. Caller will get the notification if certain resource is changed.
EFI Protocol

```c
typedef EFI_STATUS
(EFIAPI *EFI_REST_EX_EVENT_SERVICE)(
    IN EFI_REST_EX_PROTOCOL *This,
    IN EFI_HTTP_MESSAGE *RequestMessage OPTIONAL,
    IN EFI_REST_EX_TOKEN *RestExToken
);
```

**Parameters**

- **This**
  This is the EFI_REST_EX_PROTOCOL instance.

- **RequestMessage**
  This is the HTTP request message sent to REST service. Set RequestMessage to NULL to cancel the previous event service associated with the corresponding RestExToken. See descriptions for the details.

- **RestExToken**
  REST EX token which REST EX Protocol driver instance uses to notify REST client the URI resource which monitored by REST client has been changed. See the related definition of `EFI_REST_EX_TOKEN` in `EFI_REST_EX_PROTOCOL.AsyncSendReceive()`.

**Description**

This function is used to subscribe an event through REST Event service if REST service supports event service. This function listens on resource change of specific REST URI resource. The type of URI resource change event is varied and REST service specific, such as URI resource updated, resource added, resource removed, alert, etc. The way to subscribe REST Event service is also REST service specific, usually described in HTTP body. With the implementation of `EFI_REST_EX_PROTOCOL.EventService()`, REST client can register an REST EX token of particular URI resource change, usually of a time critical nature, until subscription is deleted from REST Event service.

The implementation of underlying mechanism of REST Event service depends on the interface of REST EX Protocol driver instance. HTTP protocol, In-Band management protocols or other protocols can have its own implementation to support REST Event Service request. REST EX Protocol driver instance has knowledge of how to handle the REST Event service. The REST client creates and submits an HTTP-like header/body content in `RequestMessage` which required by REST Event services. How does REST EX Protocol driver instance handle REST Event service and monitor event is REST service-specific. REST EX driver instance can just returns `EFI_UNSUPPORTED` if REST service has no event capability. Also, this function must returns `EFI_UNSUPPORTED` if `EFI_REST_EX_SERVICE_TYPE` returned in `EFI_REST_EX_SERVICE_INFO` from GetService() is `EFI_REST_EX_SERVICE_UNSPECIFIC`.

The REST EX Protocol driver instance is responsible to monitor the resource change event pushed from REST service. REST EX Protocol driver instance signals caller’s `RestExToken` when the event of resource change is pushed to REST EX Protocol driver instance. The way how REST service pushes event to REST EX Protocol driver instance is implementation-specific and transparent to REST client. REST EX Protocol driver instance sets Status field in `RestExToken` to `EFI_SUCCESS` and sets `ResponseMessage` pointer to the event resource returned from REST Event service. Then REST EX Protocol driver instance signals caller’s event to notify REST client a new REST event is received. REST EX Protocol driver instance also responsible to terminate event subscription and clear up the internal resource associated with REST Event service if the status of subscription resource is returned error.
REST EX Protocol driver instance must has capability to remove event subscription created by REST client. Caller has to set RequestMessage to NULL with RestExToken set to `EFI_REST_EX_TOKEN` which was successfully sent to this function previously. REST EX Protocol driver instance finds the given `EFI_REST_EX_TOKEN` from its private database and delete the associated event from REST service.

**Status Code Returned**

<table>
<thead>
<tr>
<th>Code</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>EFI_SUCCESS</td>
<td>Asynchronous REST request is established.</td>
</tr>
<tr>
<td>EFI_UNSUPPORTED</td>
<td>This REST EX Protocol driver instance doesn’t support asynchronous request.</td>
</tr>
<tr>
<td>EFI_ABORT</td>
<td>Previous asynchronous REST request has been canceled or event subscription has been delete from service.</td>
</tr>
<tr>
<td>EFI_DEVICE_ERROR</td>
<td>Otherwise, returns EFI_DEVICE_ERROR for other errors according to HTTP Status Code.</td>
</tr>
<tr>
<td>EFI_NOT_READY</td>
<td>The configuration of this instance is not set yet. Configure() must be executed and returns successfully prior to invoke this function</td>
</tr>
</tbody>
</table>

### 29.7.2.3 Usage Example (HTTP-aware REST EX Protocol Driver Instance)

The following code example shows how a consumer of REST EX driver would use EFI REST EX ServiceBinding Protocol and EFI REST EX Protocol to send and receive the resources from a REST service.

```c
EFI_HANDLE ImageHandle;
EFI_HANDLE *HandleBuffer;
UINTN HandleNum;
UINTN Index;
EFI_REST_EX_SERVICE_BINDING_PROTOCOL *RestExService;
EFI_HANDLE RestExChild;
EFI_HANDLE *RestEx;
EFI_REST_EX_SERVICE_INFO *RestExServiceInfo;
EFI_REST_EX_CONFIG_DATA RestExConfigData;
EFI_HTTP_MESSAGE RequestMessage;
EFI_HTTP_MESSAGE ResponseMessage;

// Locate all the handles with RESTEX ServiceBinding Protocol.
//
Status = gBS->LocateHandleBuffer (  
    ByProtocol,  
    &gEfiRestExServiceBindingProtocolGuid,  
    NULL,  
    &HandleNum,  
    &HandleBuffer  
);  
if (EFI_ERROR (Status) || (HandleNum == 0)) {  
    return EFI_ABORTED;
}  
```
for (Index = 0; Index < HandleNum; Index++) {
    //
    // Get the RESTEX ServiceBinding Protocol
    //
    Status = gBS->OpenProtocol (HandleBuffer[Index],
        &gEfiRestExServiceBindingProtocolGuid,
        (VOID **) &RestExService,
        ImageHandle,
        NULL,
        EFI_OPEN_PROTOCOL_GET_PROTOCOL);
    if (EFI_ERROR (Status)) {
        return Status;
    }
    //
    // Create the corresponding REST EX child
    //
    Status = RestExService->CreateChild (RestExService, &RestExChild);
    if (EFI_ERROR (Status)) {
        return Status;
    }
    //
    // Retrieve the REST EX Protocol from child handle
    //
    Status = gBS->OpenProtocol (RestExChild,
        &gEfiRestExProtocolGuid,
        (VOID **) &RestEx,
        ImageHandle,
        NULL,
        EFI_OPEN_PROTOCOL_GET_PROTOCOL);
    if (EFI_ERROR (Status)) {
        goto ON_EXIT;
    }
    //
    // Get the information of REST service provided by this EFI REST EX driver
    //
    Status = RestEx->GetService (RestEx,
        &RestExServiceInfo);
    if (EFI_ERROR (Status)) {
        return Status;
    }
}

ON_EXIT:
goto ON_EXIT;
}

// Check whether this REST EX service is preferred by consumer:
// 1. RestServiceAccessMode is EFI_REST_EX_SERVICE_OUT_OF_BAND_ACCESS.
// 2. RestServiceType is EFI_REST_EX_SERVICE_REDFISH.
// 3. RestExConfigType is EFI_REST_EX_CONFIG_TYPE_HTTP.
//
// if ((RestExServiceInfo->ReFiRestExServiceInfoV10.estServiceAccessMode ==
//        EFI_REST_EX_SERVICE_OUT_OF_BAND_ACCESS &&
//        RestExServiceInfo->EfiRestExServiceInfoV10.RestServiceType ==
//        EFI_REST_EX_SERVICE_REDFISH &&
//        RestExServiceInfo->EfiRestExServiceInfoV10.RestExConfigType ==
//        EFI_REST_EX_CONFIG_TYPE_HTTP) {
//    break;
// }
//
// Make sure we have found the preferred REST EX driver.
//
// if (Index == HandleNum) {
//    goto ON_EXIT;
// }

// Configure the RESTEX instance.
//
// Status = RestEx->Configure ( RestEx,
//    RestExConfigData
// );
//
// if (EFI_ERROR (Status)) {
//    goto ON_EXIT;
// }

// Send and receive the resources from a REST service.
//
// Status = RestEx->SendReceive ( RestEx,
//    &RequestMessage,
//    &ResponseMessage
// );
//
// if (EFI_ERROR (Status)) {
//    goto ON_EXIT;
// }
29.7.2.3.1 EFI_REST_EX_PROTOCOL.AsyncSendReceive()

To those HTTP-aware underlying mechanisms of the REST EX Protocol driver instance and “respond-async” prefer header aware REST service, REST EX Protocol driver instance adds additional HTTP Prefer header field (Refer to IETF RFC7240) which is set to “respond-async” in the RequestMessage. HTTP 202 Accepted Status Code is returned from REST service which indicates the REST request is accepted by REST service, however, the final result is left unknown. The way how REST service returns final response to REST EX Protocol driver instance is REST service implementation-specific and transparent to the REST client. Whether or not the REST service has a proper response to “respond-async” is REST service implementation-specific. AsyncSendReceive() must returns EFI_UNSUPPORTED if the REST service that the REST EX instance communicates with is incapable of asynchronous response.

REST EX Protocol driver instance must returns EFI_SUCCESS to caller once it gets HTTP 202 Accepted Status Code from REST service. The HTTP Location header field can be returned in HTTP 202 Accepted Status Code. REST EX Protocol driver instance may create an EFI timer to poll the status of URI returned in HTTP Location header field. The content of URI which pointed by HTTP Location header is REST service implementation-specific and not defined in REST EX Protocol specification. REST EX Protocol driver instance provider should have knowledge about how to poll the status of returning resource from given HTTP Location header.
The following flowchart describes the flow of establishing asynchronous REST request on HTTP-aware infrastructure:

Once the asynchronous REST request is established, REST EX Protocol driver instance starts to poll the status of final response on the URI returned in HTTP Location header in HTTP 202 Accepted Status code.
29.7.2.3.2 **EFI_REST_EX_PROTOCOL.EventService()**

The REST client creates and submits an HTTP-like header/body content in RequestMessage which are required by REST Event services. The REST Event Service will return an HTTP 201 (CREATED) and the Location header in the response shall contain a URI giving the location of newly created subscription resource.

The following flowchart describes the flow of subscribing to a REST Event service on HTTP-aware infrastructure:

![Flowchart](image)

Once the REST request is submitted successfully and REST EX Protocol driver instance gets the HTTP 201, REST EX Protocol driver instance starts to monitor whether resource event change is pushed to REST EX Protocol driver instance from REST service.
**EFI_REST_EX_PROTOCOL.EventQueueService()**

The REST client creates and submits an HTTP-like header/body content in RequestMessage which are required by REST Event services. The REST Event Service will return an HTTP 201 (CREATED) and the Location header in the response shall contain a URI giving the location of newly created subscription resource.

The following flowchart describes the flow of subscribing to a REST Event service on HTTP-aware infrastructure:
Once the REST request is submitted successfully and REST EX Protocol driver instance gets the HTTP 201, REST EX Protocol driver instance starts to monitor whether resource event change is pushed to REST EX Protocol driver instance from REST service.
29.7.3 EFI REST JSON Resource to C Structure Converter

29.7.3.1 Overview

EFI REST JSON Structure Protocol is designed as the centralized REST JSON Resource IN-Structure OUT (JSON-IN Structure-OUT in short) and vice versa converter for EFI REST client drivers or EFI REST client applications. This protocol provides the registration function which is invoked by upper layer EFI driver to register converter as the plug-in converter for the well-known REST JSON resource. The EFI driver which provide REST JSON resource to structure converter is EFI REST JSON structure converter producer. In the other hand, EFI drivers or applications which utilize EFI REST JSON Structure protocol is the consumer of EFI REST JSON structure converter. The convert producer is required to register its converter functions with predefined REST JSON resource namespace and data type. EFI REST JSON Structure Protocol maintains the database of all plug-in converter and dispatches the consumer request to proper REST JSON resource structure converter. EFI REST JSON Structure Protocol doesn’t have knowledge about the exact structure for the particular REST JSON resource. It just dispatches JSON resource to the correct convert functions and returns the pointer of structure generated by convert producer. This protocol reduces the burdens of JSON resource parsing effort. This also provides the easier way to refer to specific REST JSON property using native C structure reference. Below figure delineates the software stack of EFI REST JSON resource to structure converter architecture.

29.7.3.2 EFI REST JSON Structure Protocol

Summary

EFI REST JSON Structure Protocol provides function to converter producer for the registration of REST JSON resource structure converter. This protocol also provides functions of JSON-IN Structure-OUT and vice versa to converter consumer.
Protocol GUID

```c
#define EFI_REST_JSON_STRUCTURE_PROTOCOL_GUID \
  { 0xa9a048f6, 0x48a0, 0x4714, {0xb7, 0xda, 0xa9, 0xad, 0x87, 0xd4, 0xda, 0xc9}}
```

Protocol Interface Structure

```c
typedef struct _EFI_REST_JSON_STRUCTURE_PROTOCOL {
  EFI_REST_JSON_STRUCTURE_REGISTER Register;
  EFI_REST_JSON_STRUCTURE_TO_STRUCTURE ToStructure;
  EFI_REST_JSON_STRUCTURE_TO_JSON ToJson;
  EFI_REST_JSON_STRUCTURE_DESTORY_STRUCTURE DestroyStructure;
} EFI_REST_JSON_STRUCTURE_PROTOCOL;
```

Parameters

- **Register**: Register REST JSON structure converter producer.
- **ToStructure**: JSON-IN Structure-OUT function.
- **ToJson**: Structure-IN JSON-OUT function.
- **DestoryStructure**: Destroy JSON structure returned from `ToStructure` function.

Description

Each plug-in JSON resource to structure converter is required to register itself into `EFI_REST_JSON_STRUCTURE_PROTOCOL`. The plug-in JSON resource to structure converter has to provide corresponding functions for `ToStructure()`, `ToJson()` and `DestroyStructure()` for the specific REST JSON resource. `EFI_REST_JSON_STRUCTURE_PROTOCOL` maintains converter producer using the JSON resource type and version information when registration. The `ToStructure()`, `ToJson()` and `DestroyStructure()` provided by `EFI_REST_JSON_STRUCTURE_PROTOCOL` is published to converter consumer for JSON-IN Structure-OUT and vice versa conversion. `EFI_REST_JSON_STRUCTURE_PROTOCOL` is responsible for dispatching consumer request to the proper converter producer.

**EFI_REST_JSON_STRUCTURE.Register ()**

Summary

This function provides REST JSON resource to structure converter registration.

Protocol Interface

```c
typedef
  EFI_STATUS
  (EFIAPI *EFI_REST_JSON_STRUCTURE_REGISTER)(
    IN EFI_REST_JSON_STRUCTURE_PROTOCOL  *This,
    IN EFI_REST_JSON_STRUCTURE_SUPPORTED  *JsonStructureSupported,
    IN EFI_REST_JSON_STRUCTURE_TO_STRUCTURE ToStructure,
    IN EFI_REST_JSON_STRUCTURE_TO_JSON ToJson,
    IN EFI_REST_JSON_STRUCTURE_DESTORY_STRUCTURE DestroyStructure)
```

Parameters

This

JsonStructureSupported

ToStructure

ToJson

DestroyStructure

Description

This function is invoked by REST JSON resource to structure converter to register JSON-IN Structure-OUT, Structure-IN JSON-OUT and destroy JSON structure functionalities. The converter producer has to correctly specify REST resource supporting information in EFI_REST_JSON_STRUCTURE_SUPPORTED. The information includes the type name, revision and data type of REST resource. Multiple REST JSON resource to structure converters may supported in one drive, refer to below related definition.

Related Description

typedef CHAR8 *EFI_REST_JSON_RESOURCE_TYPE_DATATYPE;

//******************************************************************************
// EFI_REST_JSON_RESOURCE_TYPE_NAMESPACE
//******************************************************************************
typedef struct _EFI_REST_JSON_RESOURCE_TYPE_NAMESPACE {
    CHAR8    *ResourceTypeName;
    CHAR8    *MajorVersion;
    CHAR8    *MinorVersion;
    CHAR8    *ErrataVersion;
} EFI_REST_JSON_RESOURCE_TYPE_NAMESPACE;

Parameters

ResourceTypeName    CHAR8 pointer to the name of this REST JSON Resource.
MajorVersion        CHAR8 pointer to the string of REST JSON Resource major version.
MinorVersion        CHAR8 pointer to the string of REST JSON Resource minor version.
ErrataVersion       CHAR8 pointer to the string of REST JSON Resource errata version.

//******************************************************************************
// EFI_REST_JSON_RESOURCE_TYPE_IDENTIFIER
//******************************************************************************
typedef struct _EFI_REST_JSON_RESOURCE_TYPE_IDENTIFIER {
    EFI_REST_JSON_RESOURCE_TYPE_NAMESPACE   Namespace;
    EFI_REST_JSON_RESOURCE_TYPE_DATATYPE   Datatype;
} EFI_REST_JSON_RESOURCE_TYPE_IDENTIFIER;

Parameters

Namespace Name space of this REST JSON resource.
Datatype **CHAR8**  pointer to the string of data type, could be NULL if there is no data type for this REST JSON resource.

```
//*******************************************************************************
// EFI_REST_JSON_STRUCTURE_SUPPORTED
//*******************************************************************************
typedef struct _EFI_REST_JSON_STRUCTURE_SUPPORTED{
  EFI_REST_JSON_STRUCTURE_SUPPORTED *Next;
  EFI_REST_JSON_RESOURCE_TYPE_IDENTIFIER JsonResourceType;
  } EFI_REST_JSON_STRUCTURE_SUPPORTED;
```

**Parameters**
- **Next**  Pointer to next `EFI_REST_JSON_STRUCTURE_SUPPORTED`.
- **JsonResourceType**  Information of REST JSON resource this converter supports.

**Status Codes Returned**

<table>
<thead>
<tr>
<th>Status Code</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>EFI_SUCCESS</code></td>
<td>Converter is successfully registered</td>
</tr>
<tr>
<td><code>EFI_INVALID_PARAMETER</code></td>
<td>One or more of the following is <strong>TRUE</strong>:</td>
</tr>
<tr>
<td></td>
<td>This is <strong>NULL</strong>.</td>
</tr>
<tr>
<td></td>
<td><code>JsonStructureSupported</code> is <strong>NULL</strong>.</td>
</tr>
<tr>
<td></td>
<td><code>ResourceTypeName</code> in <code>JsonStructureSupported</code> structure is a <strong>NULL</strong> string</td>
</tr>
<tr>
<td></td>
<td><code>ToStructure</code> is <strong>NULL</strong>.</td>
</tr>
<tr>
<td></td>
<td><code>ToJason</code> is <strong>NULL</strong>.</td>
</tr>
<tr>
<td></td>
<td><code>DestroyStructure</code> is <strong>NULL</strong>.</td>
</tr>
<tr>
<td><code>EFI_ALREADY_STARTED</code></td>
<td>If the JSON resource to structure converter is already registered for this</td>
</tr>
<tr>
<td></td>
<td>type and revision of JSON resource.</td>
</tr>
<tr>
<td><code>EFI_OUT_OF_RESOURCE</code></td>
<td>Not enough resource for the converter registration</td>
</tr>
</tbody>
</table>

**EFI_REST_JSON_STRUCTURE.ToStructure ()**

**Summary**

JSON-IN Structure-OUT function. Convert the given REST JSON resource into structure.

**Protocol Interface**

```
typedef
  EFI_STATUS
  (EFIAPI *EFI_REST_JSON_STRUCTURE_TO_STRUCTURE)(
    IN   EFI_REST_JSON_STRUCTURE_PROTOCOL   *This,
    IN   EFI_REST_JSON_RESOURCE_TYPE_IDENTIFIER *JsonSrcIdentifier
    OPTIONAL,
    IN   CHAR8                               *ResourceJsonText,
    OUT  EFI_REST_JSON_STRUCTURE_HEADER      **JsonStructure
  )
```

UEFI Forum, Inc.  March 2021  1579
Parameters

- **This**: This is the **EFI_REST_JSON_STRUCTURE_PROTOCOL** instance.
- **JsonRsncIdentifier**: This indicates the resource type and version is given in **ResourceJsonText**. If **JsonRsncIdentifier** is NULL, means the JSON resource type and version information of given **ResourceJsonText** is unsure. User would like to have **EFI_REST_JSON_STRUCTURE_PROTOCOL** to look for the proper JSON structure converter.
- **ResourceJsonText**: REST JSON resource in text format.
- **JsonStructure**: Pointer to receive the pointer to **EFI_REST_JSON_STRUCTURE_HEADER**, refer to related definition for the details.

Description

This function converts the given JSON resource in text format into predefined structure. The definition of structure format is not the scope of **EFI_REST_JSON_STRUCTURE_PROTOCOL**. **EFI_REST_JSON_STRUCTURE_PROTOCOL** is a centralized JSON-IN Structure-OUT converter which maintain the registration of a variety of JSON resource to structure converters. The structure definition and the corresponding C header file are written and released by 3rd party, OEM, organization or any open source communities. The JSON resource to structure converter (convert producer) may be released in the source format or binary format. The convert producer registers itself to **EFI_REST_JSON_STRUCTURE_PROTOCOL** uses Register() and provides EFI JSON resource to structure and vice versa conversion. Consumer has to destroy **JsonStructure** using DestroyStructure() function. Resource allocated for **JsonStructure** will be released and cleaned up by converter producer.

When **JsonRsncIdentifier** is a non NULL pointer, **ResourceTypeName** in **EFI_REST_JSON_RESOURCE_TYPE_NAMESPACE** must be a non NULL string, however the revision in **EFI_REST_JSON_RESOURCE_TYPE_NAMESPACE** and data type in **EFI_REST_JSON_RESOURCE_TYPE** could be NULL string if REST JSON resource is non version controlled or no data type is defined. If **JsonRsncIdentifier** is a non NULL pointer, **EFI_REST_JSON_STRUCTURE_PROTOCOL** looks for the proper converter from its database. Invokes the ToStructure() provided by the converter to convert JSON resource to structure.

Another scenario is **JsonRsncIdentifier** may passed in as NULL, this means the JSON resource type and version information of given **ResourceJsonText** is unsure. In this case, **EFI_REST_JSON_STRUCTURE_PROTOCOL** invokes and passes **ResourceJsonText** to ToStructure() of each registered converter with **JsonRsncIdentifier** set to NULL. Converter producer may or may not automatically determine REST JSON resource type and version. Converter producer should return **EFI_UNSUPPORTED** if it doesn’t support automatically recognition of REST JSON resource. Or converter producer can recognize the given REST JSON resource by parsing the certain properties. This depends on the implementation of JSON resource to structure converter. If one of the registered converter producers can recognize the given **ResourceJsonText**, the **JsonRsncIdentifier** in **EFI_REST_JSON_STRUCTURE_HEADER** is filled up with the proper REST JSON resource type, version and data type. With the information provided in **EFI_REST_JSON_STRUCTURE_HEADER**, consumer has idea about what the exact type of REST JSON structure is.
Related Description

```c
typedef struct _EFI_REST_JSON_STRUCTURE_HEADER {
  EFI_REST_JSON_RESOURCE_TYPE_IDENTIFIER JsonRsrcIdentifier;
  // Follow by a pointer points to JSON structure, the content in the
  // JSON structure is implementation-specific according to converter
  // producer.
  // VOID                                    *JsonStructurePointer;
} EFI_REST_JSON_STRUCTURE_HEADER;
```

Parameters

- **JsonRsrcIdentifier**: Information of REST JSON structure returned from this converter.
- **JsonStructurePointer**: Pointers to JSON structure, the content in the JSON structure is implementation-specific according to the converter producer.

Status Codes Returned

<table>
<thead>
<tr>
<th>Status Code</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>EFI_SUCCESS</td>
<td>Pointer to JSON structure is returned in <code>JsonStructure</code></td>
</tr>
<tr>
<td>EFI_INVALID_PARAMETER</td>
<td>One or more of the following is <code>TRUE</code>:</td>
</tr>
<tr>
<td></td>
<td>This is <code>NULL</code>.</td>
</tr>
<tr>
<td></td>
<td>ResourceJsonText is <code>NULL</code>.</td>
</tr>
<tr>
<td></td>
<td>JsonRsrcIdentifier is not <code>NULL</code>, but the ResourceTypeName in JsonRsrcIdentifier is <code>NULL</code>.</td>
</tr>
<tr>
<td></td>
<td>JsonStructure is <code>NULL</code>.</td>
</tr>
<tr>
<td>EFI_NOT_FOUND</td>
<td>No proper JSON resource to structure convert found.</td>
</tr>
</tbody>
</table>

**EFI_REST_JSON_STRUCTURE.ToJson ()**

Summary

Structure-IN JSON-OUT function. Convert the given REST JSON structure into JSON text. The definition of structure format is not the scope of **EFI_REST_JSON_STRUCTURE_PROTOCOL**. The structure definition and the corresponding C header file are written and released by 3rd party, OEM, organization or any open source communities. Consumer has to free the memory block allocated for ResourceJsonText if the JSON resource is no longer needed.
Protocol Interface

typedef
EFI_STATUS
(EFI_API *EFI_REST_JSON_STRUCTURE_TO_JSON)(
    IN EFI_REST_JSON_STRUCTURE_PROTOCOL *This,
    IN EFI_REST_JSON_STRUCTURE_HEADER *JsonStructureHeader,
    OUTCHAR8 **ResourceJsonText
);

Parameters

This This is the EFI_REST_JSON_STRUCTURE_PROTOCOL instance.

JsonStructureHeader The point to EFI_REST_JSON_STRUCTURE_HEADER structure. The
EFI_REST_JSON_RESOURCE_TYPE_IDENTIFIER in
EFI_REST_JSON_STRUCTURE_HEADER must exactly describes the
JSON resource type and revision referred by this JSON structure.

ResourceTypeName in JsonRsrcIdentifier must be non NULL
pointer points to string. Revision and data type in
JsonRsrcIdentifier could be NULL if REST JSON resource is not
version controlled and or data type definition.

ResourceJsonText Pointer to receive REST JSON resource in text format.

Description

This functions converts the given REST JSON structure into REST JSON text format resource.

Status Codes Returned

<table>
<thead>
<tr>
<th>Status Code</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>EFI_SUCCESS</td>
<td>Pointer to JSON resource in text format is returned in ResourceJsonText.</td>
</tr>
<tr>
<td>EFI_INVALID_PARAMETER</td>
<td>One or more of the following is <strong>TRUE</strong>: This is NULL. JsonStructureHeader is NULL. ResourceJsonText is NULL.</td>
</tr>
<tr>
<td>EFI_NOT_FOUND</td>
<td>No proper JSON structure convert found to convert JSON structure to JSON text format.</td>
</tr>
</tbody>
</table>
**EFI_REST_JSON_STRUCTURE.DestroyStructure ()**

**Summary**
This function destroys the REST JSON structure.

**Protocol Interface**
```c
typedef EFI_STATUS (EFIAPIC *EFI_REST_JSON_STRUCTURE_DESTORY_STRUCTURE)(
   IN EFI_REST_JSON_STRUCTURE_PROTOCOL *This,
   IN EFI_REST_JSON_STRUCTURE_HEADER *JsonStructureHeader
);
```

**Description**
This function destroys the JSON structure generated by `ToStructure()` function. REST JSON resource structure converter is responsible for freeing and cleaning up all resource associated with the give JSON structure.

**Status Codes Returned**

<table>
<thead>
<tr>
<th>Code</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>EFI_SUCCESS</td>
<td>JSON structure is successfully destroyed.</td>
</tr>
</tbody>
</table>
| EFI_INVALID_PARAMETER | One or more of the following is `TRUE`:
|                     |   · This is `NULL`.                                                        |
|                     |   · JsonStructureHeader is `NULL`.                                         |
| EFI_NOT_FOUND       | No proper JSON structure converter found to destroy JSON structure.       |

**29.7.3.3 EFI Redfish JSON Structure Converter**
Refer to Section 31.2 for writing and using an EFI Redfish JSON Structure Converter, using the `EFI_REST_JSON_STRUCTURE_PROTOCOL` protocol.
30 - Network Protocols — UDP and MTFTP

30.1 EFI UDP Protocol

This chapter defines the EFI UDP (User Datagram Protocol) Protocol that interfaces over the EFI IP Protocol, and the EFI MTFTP Protocol interface that is built upon the EFI UDP Protocol. Protocols for version 4 and version 6 of UDP and MTFTP are included.

30.1.1 UDP4 Service Binding Protocol

EFI_UDP4_SERVICE_BINDING_PROTOCOL

Summary
The EFI UDPv4 Service Binding Protocol is used to locate communication devices that are supported by an EFI UDPv4 Protocol driver and to create and destroy instances of the EFI UDPv4 Protocol child protocol driver that can use the underlying communications device.

GUID
#define EFI_UDP4_SERVICE_BINDING_PROTOCOL_GUID \
  {0x83f01464,0x99bd,0x45e5,\ 
   {0xb3,0x83,0xaf,0x63,0x05,0xd8,0xe9,0xe6}}

Description
A network application that requires basic UDPv4 I/O services can use one of the protocol handler services, such as BS->LocateHandleBuffer(), to search for devices that publish a EFI UDPv4 Service Binding Protocol GUID. Each device with a published EFI UDPv4 Service Binding Protocol GUID supports the EFI UDPv4 Protocol and may be available for use.

After a successful call to the EFI_UDP4_SERVICE_BINDING_PROTOCOL.CreateChild() function, the newly created child EFI UDPv4 Protocol driver is in an unconfigured state; it is not ready to send and receive data packets.

Before a network application terminates execution every successful call to the EFI_UDP4_SERVICE_BINDING_PROTOCOL.CreateChild() function must be matched with a call to the EFI_UDP4_SERVICE_BINDING_PROTOCOL.DestroyChild() function.

30.1.2 UDP4 Protocol

EFI_UDP4_PROTOCOL

Summary
The EFI UDPv4 Protocol provides simple packet-oriented services to transmit and receive UDP packets.
GUID

```c
#define EFI_UDP4_PROTOCOL_GUID \
{0x3ad9df29,0x4501,0x478d,\ 
{0xb1,0xf8,0x7f,0x7f,0xe7,0x0e,0x50,0xf3}}
```

Protocol Interface Structure

```c
typedef struct _EFI_UDP4_PROTOCOL {
    EFI_UDP4_GET_MODE_DATA    GetModeData;
    EFI_UDP4_CONFIGURE        Configure;
    EFI_UDP4_GROUPS           Groups;
    EFI_UDP4_ROUTES           Routes;
    EFI_UDP4_TRANSMIT         Transmit;
    EFI_UDP4_RECEIVE          Receive;
    EFI_UDP4_CANCEL           Cancel;
    EFI_UDP4_POLL             Poll;
} EFI_UDP4_PROTOCOL;
```

Parameters

- **GetModeData**: Reads the current operational settings. See the `GetModeData()` function description.
- **Configure**: Initializes, changes, or resets operational settings for the EFI UDPv4 Protocol. See the `Configure()` function description.
- **Groups**: Joins and leaves multicast groups. See the `Groups()` function description.
- **Routes**: Add and deletes routing table entries. See the `Routes()` function description.
- **Transmit**: Queues outgoing data packets into the transmit queue. This function is a nonblocked operation. See the `Transmit()` function description.
- **Receive**: Places a receiving request token into the receiving queue. This function is a nonblocked operation. See the `Receive()` function description.
- **Cancel**: Aborts a pending transmit or receive request. See the `Cancel()` function description.
- **Poll**: Polls for incoming data packets and processes outgoing data packets. See the `Poll()` function description.

Description

The `EFI_UDP4_PROTOCOL` defines an EFI UDPv4 Protocol session that can be used by any network drivers, applications, or daemons to transmit or receive UDP packets. This protocol instance can either be bound to a specified port as a service or connected to some remote peer as an active client. Each instance has its own settings, such as the routing table and group table, which are independent from each other.
Note: In this document, all IPv4 addresses and incoming/outgoing packets are stored in network byte order. All other parameters in the functions and data structures that are defined in this document are stored in host byte order.

**EFI_UDP4_PROTOCOL.GetModeData()**

**Summary**

Reads the current operational settings.

** Prototype **

```c
typedef
EFI_STATUS
(EFI_API *EFI_UDP4_GET_MODE_DATA) (  
    IN EFI_UDP4_PROTOCOL *This,  
    OUT EFI_UDP4_CONFIG_DATA *Udp4ConfigData OPTIONAL,  
    OUT EFI_IP4_MODE_DATA *Ip4ModeData OPTIONAL,  
    OUT EFI_MANAGED_NETWORK_CONFIG_DATA *MnpConfigData OPTIONAL,  
    OUT EFI_SIMPLE_NETWORK_MODE *SnpModeData OPTIONAL
);
```

** Parameters **

- `This`: Pointer to the ` EFI_UDP4_PROTOCOL ` instance.
- `Udp4ConfigData`: Pointer to the buffer to receive the current configuration data. Type ` EFI_UDP4_CONFIG_DATA ` is defined in “Related Definitions” below.
- `Ip4ModeData`: Pointer to the EFI IPv4 Protocol mode data structure. Type ` EFI_IP4_MODE_DATA ` is defined in ` EFI_IP4_PROTOCOL.GetModeData() `. 
- `MnpConfigData`: Pointer to the managed network configuration data structure. Type ` EFI_MANAGED_NETWORK_CONFIG_DATA ` is defined in ` EFI_MANAGED_NETWORK_PROTOCOL.GetModeData() `. 
- `SnpModeData`: Pointer to the simple network mode data structure. Type ` EFI_SIMPLE_NETWORK_MODE ` is defined in the ` EFI_SIMPLE_NETWORK_PROTOCOL `. 

** Description **

The `GetModeData()` function copies the current operational settings of this EFI UDPv4 Protocol instance into user-supplied buffers. This function is used optionally to retrieve the operational mode data of underlying networks or drivers.
Related Definition

```c
typedef struct {
    //Receiving Filters
    BOOLEAN AcceptBroadcast;
    BOOLEAN AcceptPromiscuous;
    BOOLEAN AcceptAnyPort;
    BOOLEAN AllowDuplicatePort;
    // I/O parameters
    UINT8 TypeOfService;
    UINT8 TimeToLive;
    BOOLEAN DoNotFragment;
    UINT32 ReceiveTimeout;
    UINT32 TransmitTimeout;
    // Access Point
    BOOLEAN UseDefaultAddress;
    EFI_IPv4_ADDRESS StationAddress;
    EFI_IPv4_ADDRESS SubnetMask;
    UINT16 StationPort;
    EFI_IPv4_ADDRESS RemoteAddress;
    UINT16 RemotePort;
} EFI_UDP4_CONFIG_DATA;
```

- **AcceptBroadcast**: Set to `TRUE` to accept broadcast UDP packets.
- **AcceptPromiscuous**: Set to `TRUE` to accept UDP packets that are sent to any address.
- **AcceptAnyPort**: Set to `TRUE` to accept UDP packets that are sent to any port.
- **AllowDuplicatePort**: Set to `TRUE` to allow this EFI UDPv4 Protocol child instance to open a port number that is already being used by another EFI UDPv4 Protocol child instance.
- **TypeOfService**: `TypeOfService` field in transmitted IPv4 packets.
- **TimeToLive**: `TimeToLive` field in transmitted IPv4 packets.
- **DoNotFragment**: Set to `TRUE` to disable IP transmit fragmentation.
- **ReceiveTimeout**: The receive timeout value (number of microseconds) to be associated with each incoming packet. Zero means do not drop incoming packets.
- **TransmitTimeout**: The transmit timeout value (number of microseconds) to be associated with each outgoing packet. Zero means do not drop outgoing packets.
- **UseDefaultAddress**: Set to `TRUE` to use the default IP address and default routing table. If the default IP address is not available yet, then the underlying EFI IPv4 Protocol driver will use
**EFI_IP4_CONFIG2_PROTOCOL** to retrieve the IP address and subnet information. Ignored for incoming filtering if **AcceptPromiscuous** is set to **TRUE**.

**StationAddress** The station IP address that will be assigned to this EFI UDIPv4 Protocol instance. The EFI UDIPv4 and EFI IPv4 Protocol drivers will only deliver incoming packets whose destination matches this IP address exactly. Address 0.0.0.0 is also accepted as a special case in which incoming packets destined to any station IP address are always delivered. Not used when **UseDefaultAddress** is **TRUE**. Ignored for incoming filtering if **AcceptPromiscuous** is **TRUE**.

**SubnetMask** The subnet address mask that is associated with the station address. Not used when **UseDefaultAddress** is **TRUE**.

**StationPort** The port number to which this EFI UDIPv4 Protocol instance is bound. If a client of the EFI UDIPv4 Protocol does not care about the port number, set **StationPort** to zero. The EFI UDIPv4 Protocol driver will assign a random port number to transmitted UDP packets. Ignored if **AcceptAnyPort** is set to **TRUE**.

**RemoteAddress** The IP address of remote host to which this EFI UDIPv4 Protocol instance is connecting. If **RemoteAddress** is not 0.0.0.0, this EFI UDIPv4 Protocol instance will be connected to **RemoteAddress**; i.e., outgoing packets of this EFI UDIPv4 Protocol instance will be sent to this address by default and only incoming packets from this address will be delivered to client. Ignored for incoming filtering if **AcceptPromiscuous** is **TRUE**.

**RemotePort** The port number of the remote host to which this EFI UDIPv4 Protocol instance is connecting. If it is not zero, outgoing packets of this EFI UDIPv4 Protocol instance will be sent to this port number by default and only incoming packets from this port will be delivered to client. Ignored if **RemoteAddress** is 0.0.0.0 and ignored for incoming filtering if **AcceptPromiscuous** is **TRUE**.

### Status Codes Returned

<table>
<thead>
<tr>
<th>Status Code</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>EFI_SUCCESS</td>
<td>The mode data was read.</td>
</tr>
<tr>
<td>EFI_NOT_STARTED</td>
<td>When <strong>Udp4ConfigData</strong> is queried, no configuration data is available because the instance has not been started.</td>
</tr>
<tr>
<td>EFI_INVALID_PARAMETER</td>
<td><strong>This</strong> is <strong>NULL</strong>.</td>
</tr>
</tbody>
</table>

**EFI_UDPV4_PROTOCOL**.**Configure()**

**Summary**

- Initializes, changes, or resets the operational parameters for this instance of the EFI UDIPv4 Protocol.
Prototype

```c
typedef EFI_STATUS
(EFIAPI *EFI_UDP4_CONFIGURE) (
    IN EFI_UDP4_PROTOCOL *This,
    IN EFI_UDP4_CONFIG_DATA *UdpConfigData OPTIONAL
);
```

Parameters

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>This</td>
<td>Pointer to the EFI_UDP4_PROTOCOL instance.</td>
</tr>
<tr>
<td>UdpConfigData</td>
<td>Pointer to the buffer to receive the current mode data.</td>
</tr>
</tbody>
</table>

Description

The **Configure()** function is used to do the following:

- Initialize and start this instance of the EFI UDPv4 Protocol.
- Change the filtering rules and operational parameters.
- Reset this instance of the EFI UDPv4 Protocol.

Until these parameters are initialized, no network traffic can be sent or received by this instance. This instance can be also reset by calling **Configure()** with **UdpConfigData** set to **NULL**. Once reset, the receiving queue and transmitting queue are flushed and no traffic is allowed through this instance.

With different parameters in **UdpConfigData**, **Configure()** can be used to bind this instance to specified port.
### Status Codes Returned

<table>
<thead>
<tr>
<th>Status Code</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>EFI_SUCCESS</td>
<td>The configuration settings were set, changed, or reset successfully.</td>
</tr>
<tr>
<td>EFI_NO_MAPPING</td>
<td>When using a default address, configuration (DHCP, BOOTP, RARP, etc.) is not finished yet.</td>
</tr>
</tbody>
</table>
| EFI_INVALID_PARAMETER| One or more following conditions are **TRUE**:  
  - **This** is **NULL**.  
  - **UdpConfigData.StationAddress** is not a valid unicast IPv4 address.  
  - **UdpConfigData.SubnetMask** is not a valid IPv4 address mask. The subnet mask must be contiguous.  
  - **UdpConfigData.RemoteAddress** is not a valid unicast IPv4 address if it is not zero. |
| EFI_ALREADY_STARTED  | The EFI UDPv4 Protocol instance is already started/configured and must be stopped/reset before it can be reconfigured. Only **TypeOfService**, **TimeToLive**, **DoNotFragment**, **ReceiveTimeout**, and **TransmitTimeout** can be reconfigured without stopping the current instance of the EFI UDPv4 Protocol. |
| EFI_ACCESS_DENIED    | **UdpConfigData.AllowDuplicatePort** is **FALSE** and **UdpConfigData.StationPort** is already used by other instance.                           |
| EFI_OUT_OF_RESOURCES | The EFI UDPv4 Protocol driver cannot allocate memory for this EFI UDPv4 Protocol instance.                                                   |
| EFI_DEVICE_ERROR     | An unexpected network or system error occurred and this instance was not opened.                                                             |

### EFI_UDP4_PROTOCOL.Groups()

**Summary**

Joins and leaves multicast groups.

**Prototype**

```c
typedef EFI_STATUS
(EIFIAPI *EFI_UDP4_GROUPS) (  
    IN EFI_UDP4_PROTOCOL *This,  
    IN BOOLEAN JoinFlag,  
    IN EFI_IPv4_ADDRESS *MulticastAddress OPTIONAL  
);
```

**Parameters**

- **This**
  - Pointer to the **EFI_UDP4_PROTOCOL** instance.

- **JoinFlag**
  - Set to **TRUE** to join a multicast group. Set to **FALSE** to leave one or all multicast groups.

- **MulticastAddress**
  - Pointer to multicast group address to join or leave.
Description

The Groups() function is used to enable and disable the multicast group filtering. If the JoinFlag is FALSE and the MulticastAddress is NULL, then all currently joined groups are left.

Status Codes Returned

<table>
<thead>
<tr>
<th>Status Code</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>EFI_SUCCESS</td>
<td>The operation completed successfully.</td>
</tr>
<tr>
<td>EFI_NOT_STARTED</td>
<td>The EFI UDPv4 Protocol instance has not been started.</td>
</tr>
<tr>
<td>EFI_NO_MAPPING</td>
<td>When using a default address, configuration (DHCP, BOOTP, RARP, etc.) is not finished yet.</td>
</tr>
<tr>
<td>EFI_OUT_OF_RESOURCES</td>
<td>Could not allocate resources to join the group.</td>
</tr>
</tbody>
</table>
| EFI_INVALID_PARAMETER | One or more of the following conditions is TRUE:  
  - This is NULL.  
  - JoinFlag is TRUE and MulticastAddress is NULL.  
  - JoinFlag is TRUE and *MulticastAddress is not a valid multicast address. |
| EFI_ALREADY_STARTED | The group address is already in the group table (when JoinFlag is TRUE). |
| EFI_NOT_FOUND | The group address is not in the group table (when JoinFlag is FALSE). |
| EFI_DEVICE_ERROR | An unexpected system or network error occurred. |

EFI_UDP4_PROTOCOL.Routes()

Summary

Adds and deletes routing table entries.

Prototype

```
typedef EFI_STATUS
  (EFIAPI *EFI_UDP4_ROUTES) (  
  IN EFI_UDP4_PROTOCOL  *This,
  IN BOOLEAN             DeleteRoute,
  IN EFI_IPv4_ADDRESS    *SubnetAddress,
  IN EFI_IPv4_ADDRESS    *SubnetMask,
  IN EFI_IPv4_ADDRESS    *GatewayAddress
  );
```

Parameters

<table>
<thead>
<tr>
<th>Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>This</td>
<td>Pointer to the EFI_UDP4_PROTOCOL instance.</td>
</tr>
<tr>
<td>DeleteRoute</td>
<td>Set to TRUE to delete this route from the routing table. Set to FALSE to add this route to the routing table.</td>
</tr>
<tr>
<td>DestinationAddress and SubnetMask</td>
<td>are used as the key to each route entry.</td>
</tr>
</tbody>
</table>
SubnetAddress | The destination network address that needs to be routed.
SubnetMask | The subnet mask of SubnetAddress.
GatewayAddress | The gateway IP address for this route.

Description
The **Routes()** function adds a route to or deletes a route from the routing table.

Routes are determined by comparing the SubnetAddress with the destination IP address and arithmetically **AND-ing** it with the SubnetMask. The gateway address must be on the same subnet as the configured station address.

The default route is added with SubnetAddress and SubnetMask both set to 0.0.0.0. The default route matches all destination IP addresses that do not match any other routes.

A zero GatewayAddress is a nonroute. Packets are sent to the destination IP address if it can be found in the Address Resolution Protocol (ARP) cache or on the local subnet. One automatic nonroute entry will be inserted into the routing table for outgoing packets that are addressed to a local subnet (gateway address of 0.0.0.0).

Each instance of the EFI UDPv4 Protocol has its own independent routing table. Instances of the EFI UDPv4 Protocol that use the default IP address will also have copies of the routing table provided by the **EFI_IP4_CONFIG2_PROTOCOL**. These copies will be updated automatically whenever the IP driver reconfigures its instances; as a result, the previous modification to these copies will be lost.

Note: There is no way to set up routes to other network interface cards (NICs) because each NIC has its own independent network stack that shares information only through **EFI UDP4 Variable**.

Status Codes Returned

<table>
<thead>
<tr>
<th>Status Code</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>EFI_SUCCESS</td>
<td>The operation completed successfully.</td>
</tr>
<tr>
<td>EFI_NOT_STARTED</td>
<td>The EFI UDPv4 Protocol instance has not been started.</td>
</tr>
<tr>
<td>EFI_NO_MAPPING</td>
<td>When using a default address, configuration (DHCP, BOOTP, RARP, etc.) is not finished yet.</td>
</tr>
<tr>
<td>EFI_INVALID_PARAMETER</td>
<td>One or more of the following conditions is True:</td>
</tr>
<tr>
<td></td>
<td>• This is <strong>NULL</strong>.</td>
</tr>
<tr>
<td></td>
<td>• SubnetAddress is <strong>NULL</strong>.</td>
</tr>
<tr>
<td></td>
<td>• SubnetMask is <strong>NULL</strong>.</td>
</tr>
<tr>
<td></td>
<td>• GatewayAddress is <strong>NULL</strong>.</td>
</tr>
<tr>
<td></td>
<td>• <em>SubnetAddress</em> is not a valid subnet address.</td>
</tr>
<tr>
<td></td>
<td>• <em>SubnetMask</em> is not a valid subnet mask.</td>
</tr>
<tr>
<td></td>
<td>• <em>GatewayAddress</em> is not a valid unicast IP address.</td>
</tr>
<tr>
<td>EFI_OUT_OF_RESOURCES</td>
<td>Could not add the entry to the routing table.</td>
</tr>
<tr>
<td>EFI_NOT_FOUND</td>
<td>This route is not in the routing table.</td>
</tr>
<tr>
<td>EFI_ACCESS_DENIED</td>
<td>The route is already defined in the routing table.</td>
</tr>
</tbody>
</table>
EFI_UDP4_PROTOCOL.Transmit()

Summary
Queues outgoing data packets into the transmit queue.

Prototype

typedef
EFI_STATUS
(EIFIAPI *EFI_UDP4_TRANSMIT) (
    IN EFI_UDP4_PROTOCOL *This,
    IN EFI_UDP4_COMPLETION_TOKEN *Token
);

Parameters
This Pointer to the EFI_UDP4_PROTOCOL instance.
Token Pointer to the completion token that will be placed into the transmit queue. Type EFI_UDP4_COMPLETION_TOKEN is defined in “Related Definitions” below.

Description
The Transmit() function places a sending request to this instance of the EFI UDPv4 Protocol, alongside the transmit data that was filled by the user. Whenever the packet in the token is sent out or some errors occur, the Token.Event will be signaled and Token.Status is updated. Providing a proper notification function and context for the event will enable the user to receive the notification and transmitting status.

Related Definitions

//***************************************************************
// EFI_UDP4_COMPLETION_TOKEN
//***************************************************************
typedef struct {
    EFI_EVENT Event;
    EFI_STATUS Status;
    union {
        EFI_UDP4_RECEIVE_DATA *RxData;
        EFI_UDP4_TRANSMIT_DATA *TxData;
    }
} EFI_UDP4_COMPLETION_TOKEN;

Event This Event will be signaled after the Status field is updated by the EFI UDv4 Protocol driver. The type of Event must be EVT_NOTIFY_SIGNAL. The Task Priority Level (TPL) of Event must be lower than or equal to TPL_CALLBACK.

Status Will be set to one of the following values:
**EFI_SUCCESS.** The receive or transmit operation completed successfully.

**EFI_ABORTED.** The receive or transmit was aborted.

**EFI_TIMEOUT.** The transmit timeout expired.

**EFI_NETWORK_UNREACHABLE.** The destination network is unreachable. RxData is set to **NULL** in this situation.

**EFI_HOST_UNREACHABLE.** The destination host is unreachable. RxData is set to **NULL** in this situation.

**EFI_PROTOCOL_UNREACHABLE.** The UDP protocol is unsupported in the remote system. RxData is set to **NULL** in this situation.

**EFI_PORT_UNREACHABLE.** No service is listening on the remote port. RxData is set to **NULL** in this situation.

**EFI_ICMP_ERROR.** Some other Internet Control Message Protocol (ICMP) error report was received. For example, packets are being sent too fast for the destination to receive them and the destination sent an ICMP source quench report. RxData is set to **NULL** in this situation.

**EFI_DEVICE_ERROR.** An unexpected system or network error occurred.

**EFI_NO_MEDIA.** There was a media error.

RxData

When this token is used for receiving, RxData is a pointer to **EFI_UDP4_RECEIVE_DATA**. Type **EFI_UDP4_RECEIVE_DATA** is defined below.

TxData

When this token is used for transmitting, TxData is a pointer to **EFI_UDP4_TRANSMIT_DATA**. Type **EFI_UDP4_TRANSMIT_DATA** is defined below.

The **EFI_UDP4_COMPLETION_TOKEN** structures are used for both transmit and receive operations. When used for transmitting, the Event and TxData fields must be filled in by the EFI UD Pv4 Protocol client. After the transmit operation completes, the Status field is updated by the EFI UD Pv4 Protocol and the Event is signaled.

- When used for receiving, only the Event field must be filled in by the EFI UD Pv4 Protocol client. After a packet is received, RxData and Status are filled in by the EFI UD Pv4 Protocol and the Event is signaled.

- The ICMP related status codes filled in Status are defined as follows:
  ```
  #define EFI_NETWORK_UNREACHABLE  EFIERR(100)
  #define EFI_HOST_UNREACHABLE    EFIERR(101)
  #define EFI_PROTOCOL_UNREACHABLE  EFIERR(102)
  #define EFI_PORT_UNREACHABLE    EFIERR(103)
  ```
//EFI_UDP4_RECEIVE_DATA

typedef struct {
    EFI_TIME      TimeStamp;
    EFI_EVENT     RecycleSignal;
    EFI_UDP4_SESSION_DATA   UdpSession;
    UINT32       DataLength;
    UINT32       FragmentCount;
    EFI_UDP4_FRAGMENT_DATA  FragmentTable[1];
} EFI_UDP4_RECEIVE_DATA;

TimeStamp
    Time when the EFI UDPv4 Protocol accepted the packet. TimeStamp is zero filled if timestamps are disabled or unsupported.

RecycleSignal
    Indicates the event to signal when the received data has been processed.

UdpSession
    The UDP session data including SourceAddress, SourcePort, DestinationAddress, and DestinationPort. Type EFI_UDP4_SESSION_DATA is defined below.

DataLength
    The sum of the fragment data length.

FragmentCount
    Number of fragments. May be zero.

FragmentTable
    Array of fragment descriptors. IP and UDP headers are included in these buffers if ConfigData.RawData is TRUE. Otherwise they are stripped. May be zero. Type EFI_UDP4_FRAGMENT_DATA is defined below.

EFI_UDP4_RECEIVE_DATA is filled by the EFI UDPv4 Protocol driver when this EFI UDPv4 Protocol instance receives an incoming packet. If there is a waiting token for incoming packets, the CompletionToken.Packet.RxData field is updated to this incoming packet and the CompletionToken.Event is signaled. The EFI UDPv4 Protocol client must signal the RecycleSignal after processing the packet.

- FragmentTable could contain multiple buffers that are not in the continuous memory locations. The EFI UDPv4 Protocol client might need to combine two or more buffers in FragmentTable to form their own protocol header.
typedef struct {
    EFI_IPv4_ADDRESS SourceAddress;
    UINT16 SourcePort;
    EFI_IPv4_ADDRESS DestinationAddress;
    UINT16 DestinationPort;
} EFI_UDP4_SESSION_DATA;

SourceAddress  Address from which this packet is sent. If this field is set to zero when sending packets, the address that is assigned in EFI_UDP4_PROTOCOL.Configure() is used.
SourcePort  Port from which this packet is sent. It is in host byte order. If this field is set to zero when sending packets, the port that is assigned in EFI_UDP4_PROTOCOL.Configure() is used. If this field is set to zero and unbound, a call to EFI_UDP4_PROTOCOL.Transmit() will fail.
DestinationAddress  Address to which this packet is sent.
DestinationPort  Port to which this packet is sent. It is in host byte order. If this field is set to zero and unconnected, the call to EFI_UDP4_PROTOCOL.Transmit() will fail.

The EFI_UDP4_SESSION_DATA is used to retrieve the settings when receiving packets or to override the existing settings of this EFI UDPv4 Protocol instance when sending packets.

typedef struct {
    UINT32 FragmentLength;
    VOID *FragmentBuffer;
} EFI_UDP4_FRAGMENT_DATA;

FragmentLength  Length of the fragment data buffer.
FragmentBuffer  Pointer to the fragment data buffer.

EFI_UDP4_FRAGMENT_DATA allows multiple receive or transmit buffers to be specified. The purpose of this structure is to avoid copying the same packet multiple times.
typedef struct {
    EFI_UDP4_SESSION_DATA  *UdpSessionData;
    EFI_IPv4_ADDRESS      *GatewayAddress;
    UINT32                DataLength;
    UINT32                FragmentCount;
    EFI_UDP4_FRAGMENT_DATA FragmentTable[1];
} EFI_UDP4_TRANSMIT_DATA;

UdpSessionData  If not NULL, the data that is used to override the transmitting settings. Type EFI_UDP4_SESSION_DATA is defined above.
GatewayAddress  The next-hop address to override the setting from the routing table.
DataLength      Sum of the fragment data length. Must not exceed the maximum UDP packet size.
FragmentCount   Number of fragments.
FragmentTable   Array of fragment descriptors. Type EFI_UDP4_FRAGMENT_DATA is defined above.

The EFI UDPv4 Protocol client must fill this data structure before sending a packet. The packet may contain multiple buffers that may be not in a continuous memory location.
### Status Codes Returned

<table>
<thead>
<tr>
<th>Code</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>EFI_SUCCESS</td>
<td>The data has been queued for transmission.</td>
</tr>
<tr>
<td>EFI_NOT_STARTED</td>
<td>This EFI UDPv4 Protocol instance has not been started.</td>
</tr>
<tr>
<td>EFI_NO_MAPPING</td>
<td>When using a default address, configuration (DHCP, BOOTP, RARP, etc.) is not finished yet.</td>
</tr>
<tr>
<td>EFI_INVALID_PARAMETER</td>
<td>One or more of the following are TRUE:</td>
</tr>
<tr>
<td></td>
<td>• This is NULL.</td>
</tr>
<tr>
<td></td>
<td>• Token is NULL.</td>
</tr>
<tr>
<td></td>
<td>• Token.Event is NULL.</td>
</tr>
<tr>
<td></td>
<td>• Token.Packet.TxData is NULL.</td>
</tr>
<tr>
<td></td>
<td>• Token.Packet.TxData.FragmentCount is zero.</td>
</tr>
<tr>
<td></td>
<td>• Token.Packet.TxData.DataLength is not equal to the sum of fragment lengths.</td>
</tr>
<tr>
<td></td>
<td>• One or more of the Token.Packet.TxData.FragmentTable[].FragmentLength fields is zero.</td>
</tr>
<tr>
<td></td>
<td>• One or more of the Token.Packet.TxData.FragmentTable[].FragmentBuffer fields is NULL.</td>
</tr>
<tr>
<td></td>
<td>• Token.Packet.TxData. GatewayAddress is not a unicast IPv4 address if it is not NULL.</td>
</tr>
<tr>
<td></td>
<td>• Token.Packet.TxData.UdpSessionData.SourceAddress is not a valid unicast IPv4 address or Token.Packet.TxData.UdpSessionData.DestinationAddress is zero if the UdpSessionData is not NULL.</td>
</tr>
<tr>
<td>EFI_ACCESS_DENIED</td>
<td>The transmit completion token with the same Token.Event was already in the transmit queue.</td>
</tr>
<tr>
<td>EFI_NOT_READY</td>
<td>The completion token could not be queued because the transmit queue is full.</td>
</tr>
<tr>
<td>EFI_OUT_OF_RESOURCES</td>
<td>Could not queue the transmit data.</td>
</tr>
<tr>
<td>EFI_NOT_FOUND</td>
<td>There is no route to the destination network or address.</td>
</tr>
<tr>
<td>EFI_BAD_BUFFER_SIZE</td>
<td>The data length is greater than the maximum UD P packet size. Or the length of the IP header + UDP header + data length is greater than MTU if DoNotFragment is TRUE.</td>
</tr>
<tr>
<td>EFI_NO_MEDIA</td>
<td>There was a media error.</td>
</tr>
</tbody>
</table>

### EFI_UDP4_PROTOCOL.Receive()

**Summary**

Places an asynchronous receive request into the receiving queue.
Prototype

```c
typedef
EFI_STATUS
(EIFIAPI *EFI_UDP4_RECEIVE) (  
  IN EFI_UDP4_PROTOCOL *This,
  IN EFI_UDP4_COMPLETION_TOKEN *Token
);
```

Parameters

- **This**: Pointer to the `EFI_UDP4_PROTOCOL` instance.
- **Token**: Pointer to a token that is associated with the receive data descriptor. Type `EFI_UDP4_COMPLETION_TOKEN` is defined in `EFI_UDP4_PROTOCOL.Transmit()`.

Description

The `Receive()` function places a completion token into the receive packet queue. This function is always asynchronous.

The caller must fill in the `Token.Event` field in the completion token, and this field cannot be `NULL`. When the receive operation completes, the EFI UDPv4 Protocol driver updates the `Token.Status` and `Token.Packet.RxData` fields and the `Token.Event` is signaled. Providing a proper notification function and context for the event will enable the user to receive the notification and receiving status. That notification function is guaranteed to not be re-entered.

Status Codes Returned

<table>
<thead>
<tr>
<th>Code</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>EFI_SUCCESS</td>
<td>The receive completion token was cached.</td>
</tr>
<tr>
<td>EFI_NOT_STARTED</td>
<td>This EFI UDPv4 Protocol instance has not been started.</td>
</tr>
<tr>
<td>EFI_NO_MAPPING</td>
<td>When using a default address, configuration (DHCP, BOOTP, RARP, etc.) is not finished yet.</td>
</tr>
<tr>
<td>EFI_INVALID_PARAMETER</td>
<td>One or more of the following conditions is <code>TRUE</code>:</td>
</tr>
<tr>
<td></td>
<td>• This is <code>NULL</code>.</td>
</tr>
<tr>
<td></td>
<td>• Token is <code>NULL</code>.</td>
</tr>
<tr>
<td></td>
<td>• Token.Event is <code>NULL</code>.</td>
</tr>
<tr>
<td>EFI_OUT_OF_RESOURCES</td>
<td>The receive completion token could not be queued due to a lack of system resources (usually memory).</td>
</tr>
<tr>
<td>EFI_DEVICE_ERROR</td>
<td>An unexpected system or network error occurred. The EFI UDPv4 Protocol instance has been reset to startup defaults.</td>
</tr>
<tr>
<td>EFI_ACCESS_DENIED</td>
<td>A receive completion token with the same <code>Token.Event</code> was already in the receive queue.</td>
</tr>
<tr>
<td>EFI_NOT_READY</td>
<td>The receive request could not be queued because the receive queue is full.</td>
</tr>
<tr>
<td>EFI_NO_MEDIA</td>
<td>There was a media error.</td>
</tr>
</tbody>
</table>
**EFI_UDP4_PROTOCOL.Cancel()**

**Summary**
Aborts an asynchronous transmit or receive request.

**Prototype**
```c
typedef EFI_STATUS (EFIAPI *EFI_UDP4_CANCEL)(
    IN EFI_UDP4_PROTOCOL *This,
    IN EFI_UDP4_COMPLETION_TOKEN *Token  OPTIONAL
);
```

**Parameters**
- **This**: Pointer to the EFI_UDP4_PROTOCOL instance.
- **Token**: Pointer to a token that has been issued by EFI_UDP4_PROTOCOL.Transmit() or EFI_UDP4_PROTOCOL.Receive(). If NULL, all pending tokens are aborted. Type EFI_UDP4_COMPLETION_TOKEN is defined in EFI_UDP4_PROTOCOL.Transmit().

**Description**
The Cancel() function is used to abort a pending transmit or receive request. If the token is in the transmit or receive request queues, after calling this function, Token.Status will be set to EFI_ABORTED and then Token.Event will be signaled. If the token is not in one of the queues, which usually means that the asynchronous operation has completed, this function will not signal the token and EFI_NOT_FOUND is returned.

**Status Codes Returned**

<table>
<thead>
<tr>
<th>Code</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>EFI_SUCCESS</td>
<td>The asynchronous I/O request was aborted and Token.Event was signaled. When Token is NULL, all pending requests are aborted and their events are signaled.</td>
</tr>
<tr>
<td>EFI_INVALID_PARAMETER</td>
<td>This is NULL.</td>
</tr>
<tr>
<td>EFI_NOT_STARTED</td>
<td>This instance has not been started.</td>
</tr>
<tr>
<td>EFI_NO_MAPPING</td>
<td>When using the default address, configuration (DHCP, BOOTP, RARP, etc.) is not finished yet.</td>
</tr>
<tr>
<td>EFI_NOT_FOUND</td>
<td>When Token is not NULL, the asynchronous I/O request was not found in the transmit or receive queue. It has either completed or was not issued by Transmit() and Receive().</td>
</tr>
</tbody>
</table>

**EFI_UDP4_PROTOCOL.Poll()**

**Summary**
Polls for incoming data packets and processes outgoing data packets.
Prototype

```c
typedef
EFI_STATUS
(EFIAPI *EFI_UDP4_POLL) (   
    IN EFI_UDP4_PROTOCOL   *This
);
```

Parameters

This Pointer to the EFI_UDP4_PROTOCOL instance.

Description

The Poll() function can be used by network drivers and applications to increase the rate that data packets are moved between the communications device and the transmit and receive queues.

In some systems, the periodic timer event in the managed network driver may not poll the underlying communications device fast enough to transmit and/or receive all data packets without missing incoming packets or dropping outgoing packets. Drivers and applications that are experiencing packet loss should try calling the Poll() function more often.

Status Codes Returned

<table>
<thead>
<tr>
<th>Status Code</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>EFI_SUCCESS</td>
<td>Incoming or outgoing data was processed.</td>
</tr>
<tr>
<td>EFI_INVALID_PARAMETER</td>
<td>This is NULL.</td>
</tr>
<tr>
<td>EFI_DEVICE_ERROR</td>
<td>An unexpected system or network error occurred.</td>
</tr>
<tr>
<td>EFI_TIMEOUT</td>
<td>Data was dropped out of the transmit and/or receive queue. Consider increasing the polling rate.</td>
</tr>
</tbody>
</table>

30.2 EFI UDPv6 Protocol

This section defines the EFI UDPv6 (User Datagram Protocol version 6) Protocol that interfaces over the EFI IPv6 Protocol.

30.2.1 UDP6 Service Binding Protocol

EFI_UDP6_SERVICE_BINDING_PROTOCOL

Summary

The EFI UDPv6 Service Binding Protocol is used to locate communication devices that are supported by an EFI UDPv6 Protocol driver and to create and destroy instances of the EFI UDPv6 Protocol child instance that uses the underlying communications device.
GUID
#define EFI_UDP6_SERVICE_BINDING_PROTOCOL_GUID \
{0x66ed4721, 0x3c98, 0x4d3e,\ 
 {0x81, 0xe3, 0xd0, 0x3d, 0xd3, 0x9a, 0x72, 0x54}}

Description
A network application that requires basic UDPv6 I/O services can use one of the protocol handler services, such as BS->LocateHandleBuffer(), to search for devices that publish a EFI UDPv6 Service Binding Protocol GUID. Each device with a published EFI UDPv6 Service Binding Protocol GUID supports the EFI UDPv6 Protocol and may be available for use.

After a successful call to the EFI_UDP6_SERVICE_BINDING_PROTOCOL.CreateChild() function, the newly created child EFI UDPv6 Protocol driver is in an un-configured state; it is not ready to send and receive data packets.

Before a network application terminates execution, every successful call to the EFI_UDP6_SERVICE_BINDING_PROTOCOL.CreateChild() function must be matched with a call to the EFI_UDP6_SERVICE_BINDING_PROTOCOL.DestroyChild() function.

30.2.2 EFI UDP6 Protocol

EFI_UDP6_PROTOCOL

Summary
The EFI UDPv6 Protocol provides simple packet-oriented services to transmit and receive UDP packets.

GUID
#define EFI_UDP6_PROTOCOL_GUID \
{0x4f948815, 0xb4b9, 0x43cb,\ 
 {0x8a, 0x33, 0x90, 0xe0, 0x60, 0xb3, 0x49, 0x55}}

Protocol Interface Structure

typedef struct _EFI_UDP6_PROTOCOL  {
 EFI_UDP6_GET_MODE_DATA GetModeData;
 EFI_UDP6_CONFIGURE Configure;
 EFI_UDP6_GROUPS Groups;
 EFI_UDP6_TRANSMIT Transmit;
 EFI_UDP6_RECEIVE Receive;
 EFI_UDP6_CANCEL Cancel;
 EFI_UDP6_POLL Poll;
 } EFI_UDP6_PROTOCOL;

Parameters
GetModeData  Reads the current operational settings. See the GetModeData() function description.
Configure

Initializes, changes, or resets operational settings for the EFI UDPv6 Protocol. See the `Configure()` function description.

Groups

Joins and leaves multicast groups. See the `Groups()` function description.

Transmit

Queues outgoing data packets into the transmit queue. This function is a non-blocked operation. See the `Transmit()` function description.

Receive

Places a receiving request token into the receiving queue. This function is a non-blocked operation. See the `Receive()` function description.

Cancel

Aborts a pending transmit or receive request. See the `Cancel()` function description.

Poll

Polls for incoming data packets and processes outgoing data packets. See the `Poll()` function description.

Description

The `EFI_UDP6_PROTOCOL` defines an EFI UDPv6 Protocol session that can be used by any network drivers, applications, or daemons to transmit or receive UDP packets. This protocol instance can either be bound to a specified port as a service or connected to some remote peer as an active client. Each instance has its own settings, such as group table, that are independent from each other.

Note:  
Byte Order: In this document, all IPv6 addresses and incoming/outgoing packets are stored in network byte order. All other parameters in the functions and data structures that are defined in this document are stored in host byte order.

`EFI_UDP6_PROTOCOL.GetModeData()`

Summary

Read the current operational settings.

Prototype

```c
typedef EFI_STATUS
(EFIAPI *EFI_UDP6_GET_MODE_DATA) ( 
    IN EFI_UDP6_PROTOCOL *This, 
    OUT EFI_UDP6_CONFIG_DATA *Udp6ConfigData OPTIONAL, 
    OUT EFI_IP6_MODE_DATA *Ip6ModeData OPTIONAL, 
    OUT EFI_MANAGED_NETWORK_CONFIG_DATA *MnpConfigData OPTIONAL, 
    OUT EFI_SIMPLE_NETWORK_MODE *SnpModeData OPTIONAL );
```

Parameters

- **This**
  Pointer to the `EFI_UDP6_PROTOCOL` instance.
- **Udp6ConfigData**
  The buffer in which the current UDP configuration data is returned. Type `EFI_UDP6_CONFIG_DATA` is defined in "Related Definitions" below.
Ip6ModeData  The buffer in which the current EFI IPv6 Protocol mode data is returned. Type EFI_IP6_MODE_DATA is defined in EFI_IP6_PROTOCOL.GetModeData().

MnpConfigData The buffer in which the current managed network configuration data is returned. Type EFI_MANAGED_NETWORK_CONFIG_DATA is defined in EFI_MANAGED_NETWORK_PROTOCOL.GetModeData().

SnpModeData The buffer in which the simple network mode data is returned. Type EFI_SIMPLE_NETWORK_MODE is defined in the EFI_SIMPLE_NETWORK Protocol.

Description
The GetModeData() function copies the current operational settings of this EFI UDPv6 Protocol instance into user-supplied buffers. This function is used optionally to retrieve the operational mode data of underlying networks or drivers.

Related Definition
***************************************************************
// EFI_UDP6_CONFIG_DATA
//
***************************************************************
typedef struct {
  //Receiving Filters
  BOOLEAN AcceptPromiscuous;
  BOOLEAN AcceptAnyPort;
  BOOLEAN AllowDuplicatePort;
  //I/O parameters
;
  UINT8  TrafficClass;
  UINT8  HopLimit;
;
  UINT32 ReceiveTimeout;
  UINT32 TransmitTimeout;
  //Access Point
  EFI_IPv6_ADDRESS StationAddress;
  UINT16  StationPort;
  EFI_IPv6_ADDRESS RemoteAddress;
  UINT16  RemotePort;
} EFI_UDP6_CONFIG_DATA;

AcceptPromiscuous  Set to TRUE to accept UDP packets that are sent to any address.
AcceptAnyPort  Set to TRUE to accept UDP packets that are sent to any port.
AllowDuplicatePort

Set to **TRUE** to allow this EFI UDPv6 Protocol child instance to open a port number that is already being used by another EFI UDPv6 Protocol child instance.

TrafficClass

TrafficClass field in transmitted IPv6 packets.

HopLimit

HopLimit field in transmitted IPv6 packets.

ReceiveTimeout

The receive timeout value (number of microseconds) to be associated with each incoming packet. Zero means do not drop incoming packets.

TransmitTimeout

The transmit timeout value (number of microseconds) to be associated with each outgoing packet. Zero means do not drop outgoing packets.

StationAddress

The station IP address that will be assigned to this EFI UDPv6 Protocol instance. The EFI UDPv6 and EFI IPv6 Protocol drivers will only deliver incoming packets whose destination matches this IP address exactly. Address 0::/128 is also accepted as a special case. Under this situation, underlying IPv6 driver is responsible for binding a source address to this EFI IPv6 protocol instance according to source address selection algorithm. Only incoming packet from the selected source address is delivered. This field can be set and changed only when the EFI IPv6 driver is transitioning from the stopped to the started states. If no address is available for selecting, the EFI IPv6 Protocol driver will use **EFI_IP6_CONFIG_PROTOCOL** to retrieve the IPv6 address.

StationPort

The port number to which this EFI UDPv6 Protocol instance is bound. If a client of the EFI UDPv6 Protocol does not care about the port number, set **StationPort** to zero. The EFI UDPv6 Protocol driver will assign a random port number to transmitted UDP packets. Ignored if **AcceptAnyPort** is **TRUE**.

RemoteAddress

The IP address of remote host to which this EFI UDPv6 Protocol instance is connecting. If **RemoteAddress** is not 0::/128, this EFI UDPv6 Protocol instance will be connected to **RemoteAddress**; i.e., outgoing packets of this EFI UDPv6 Protocol instance will be sent to this address by default and only incoming packets from this address will be delivered to client. Ignored for incoming filtering if **AcceptPromiscuous** is **TRUE**.

RemotePort

The port number of the remote host to which this EFI UDPv6 Protocol instance is connecting. If it is not zero, outgoing packets of this EFI UDPv6 Protocol instance will be sent to this port number by default and only incoming packets from this port will be delivered to client. Ignored if **RemoteAddress** is 0::/128 and ignored for incoming filtering if **AcceptPromiscuous** is **TRUE**.
Status Codes Returned

<table>
<thead>
<tr>
<th>Code</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>EFI_SUCCESS</td>
<td>The mode data was read.</td>
</tr>
<tr>
<td>EFI_NOT_STARTED</td>
<td>When <code>Udp6ConfigData</code> is queried, no configuration data is available because this instance has not been started.</td>
</tr>
<tr>
<td>EFI_INVALID_PARAMETER</td>
<td><em>This</em> is NULL.</td>
</tr>
</tbody>
</table>

**EFI_UDP6_PROTOCOL.Configure()**

**Summary**
Initializes, changes, or resets the operational parameters for this instance of the EFI UDPv6 Protocol.

**Prototype**
```c
typedef EFI_STATUS (EFIAPI *EFI_UDP6_CONFIGURE) (
    IN EFI_UDP6_PROTOCOL *This,
    IN EFI_UDP6_CONFIG_DATA *UdpConfigData OPTIONAL
);
```

**Parameters**
- **This** Pointer to the `EFI_UDP6_PROTOCOL` instance.
- **UdpConfigData** Pointer to the buffer contained the configuration data.

**Description**
The `Configure()` function is used to do the following:

- Initialize and start this instance of the EFI UDPv6 Protocol.
- Change the filtering rules and operational parameters.
- Reset this instance of the EFI UDPv6 Protocol.

Until these parameters are initialized, no network traffic can be sent or received by this instance. This instance can be also reset by calling `Configure()` with `UdpConfigData` set to NULL. Once reset, the receiving queue and transmitting queue are flushed and no traffic is allowed through this instance.

With different parameters in `UdpConfigData`, `Configure()` can be used to bind this instance to specified port.
Status Codes Returned

<table>
<thead>
<tr>
<th>Code</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>EFI_SUCCESS</td>
<td>The configuration settings were set, changed, or reset successfully.</td>
</tr>
<tr>
<td>EFI_NO_MAPPING</td>
<td>The underlying IPv6 driver was responsible for choosing a source address for this instance, but no source address was available for use.</td>
</tr>
</tbody>
</table>
| EFI_INVALID_PARAMETER | One or more following conditions are **TRUE**.  
                      **This** is **NULL**  
                      UdpConfigData.StationAddress neither zero nor one of the configured IP addresses in the underlying IPv6 driver.  
                      UdpConfigData.RemoteAddress is not a valid unicast IPv6 address if it is not zero. |
| EFI_ALREADY_STARTED | The EFI UDPv6 Protocol instance is already started/configured and must be stopped/reset before it can be reconfigured. Only TrafficClass, HopLimit, ReceiveTimeout, and TransmitTimeout can be reconfigured without stopping the current instance of the EFI UDPv6 Protocol. |
| EFI_ACCESS_DENIED | UdpConfigData. AllowDuplicatePort is **FALSE** and UdpConfigData.StationPort is already used by other instance. |
| EFI_OUT_OF_RESOURCES | The EFI UDPv6 Protocol driver cannot allocate memory for this EFI UDPv6 Protocol instance. |
| EFI_DEVICE_ERROR | An unexpected network or system error occurred and this instance was not opened. |

**EFI_UDP6_PROTOCOL.Groups()**

**Summary**
Joins and leaves multicast groups.

**Prototype**
```
typedef
EFI_STATUS
(EIFIAPI *EFI_UDP6_GROUPS) (  
    IN EFI_UDP6_PROTOCOL *This,  
    IN BOOLEAN JoinFlag,  
    IN EFI_IPv6_ADDRESS *MulticastAddress OPTIONAL
);
```

**Parameters**
- **This**  
  Pointer to the **EFI_UDP6_PROTOCOL** instance.
- **JoinFlag**  
  Set to **TRUE** to join a multicast group. Set to **FALSE** to leave one or all multicast groups.
- **MulticastAddress**  
  Pointer to multicast group address to join or leave.
Description

The **Groups()** function is used to join or leave one or more multicast group.

If the **JoinFlag** is **FALSE** and the **MulticastAddress** is **NULL**, then all currently joined groups are left.

Status Codes Returned

<table>
<thead>
<tr>
<th>Code</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>EFI_SUCCESS</td>
<td>The operation completed successfully.</td>
</tr>
<tr>
<td>EFI_NOT_STARTED</td>
<td>The EFI UDPv6 Protocol instance has not been started.</td>
</tr>
<tr>
<td>EFI_OUT_OF_RESOURCES</td>
<td>Could not allocate resources to join the group.</td>
</tr>
<tr>
<td>EFI_INVALID_PARAMETER</td>
<td>One or more of the following conditions is <strong>TRUE</strong>:</td>
</tr>
<tr>
<td></td>
<td><strong>This</strong> is <strong>NULL</strong>.</td>
</tr>
<tr>
<td></td>
<td><strong>JoinFlag</strong> is <strong>TRUE</strong> and <strong>MulticastAddress</strong> is <strong>NULL</strong>.</td>
</tr>
<tr>
<td></td>
<td><strong>JoinFlag</strong> is <strong>TRUE</strong> and *<strong>MulticastAddress</strong> is not a valid multicast address.</td>
</tr>
<tr>
<td>EFI_ALREADY_STARTED</td>
<td>The group address is already in the group table (when <strong>JoinFlag</strong> is <strong>TRUE</strong>).</td>
</tr>
<tr>
<td>EFI_NOT_FOUND</td>
<td>The group address is not in the group table (when <strong>JoinFlag</strong> is <strong>FALSE</strong>).</td>
</tr>
<tr>
<td>EFI_DEVICE_ERROR</td>
<td>An unexpected system or network error occurred.</td>
</tr>
</tbody>
</table>

**EFI_UDP6_PROTOCOL.Transmit()**

**Summary**

Queues outgoing data packets into the transmit queue.

**Prototype**

```c
typedef
EFI_STATUS
(EIFIAPI *EFI_UDP6_TRANSMIT) (  
    IN EFI_UDP6_PROTOCOL    *This,
    IN EFI_UDP6_COMPLETION_TOKEN *Token
);
```

**Parameters**

- **This**  
  Pointer to the **EFI_UDP6_PROTOCOL** instance.

- **Token**  
  Pointer to the completion token that will be placed into the transmit queue. Type **EFI_UDP6_COMPLETION_TOKEN** is defined in "Related Definitions" below.

**Description**

The **Transmit()** function places a sending request to this instance of the EFI UDPv6 Protocol, alongside the transmit data that was filled by the user. Whenever the packet in the token is sent out or some errors occur, the **Token.Event** will be signaled and **Token.Status** is updated. Providing a proper notification function and context for the event will enable the user to receive the notification and transmitting status.
Related Definitions

```c
typedef struct {
    EFI_EVENT Event;
    EFI_STATUS Status;
    union {
        EFI_UDP6_RECEIVE_DATA *RxData;
        EFI_UDP6_TRANSMIT_DATA *TxData;
    }
} EFI_UDP6_COMPLETION_TOKEN;
```

- **Event**: This event will be signaled after the Status field is updated by the EFI UDPv6 Protocol driver. The type of Event must be `EVT_NOTIFY_SIGNAL`.
- **Status**: Will be set to one of the following values:
  - `EFI_SUCCESS`: The receive or transmit operation completed successfully.
  - `EFI_ABORTED`: The receive or transmit was aborted.
  - `EFI_TIMEOUT`: The transmit timeout expired.
  - `EFI_NETWORK_UNREACHABLE`: The destination network is unreachable. `RxData` is set to `NULL` in this situation.
  - `EFI_HOST_UNREACHABLE`: The destination host is unreachable. `RxData` is set to `NULL` in this situation.
  - `EFI_PROTOCOL_UNREACHABLE`: The UDP protocol is unsupported in the remote system. `RxData` is set to `NULL` in this situation.
  - `EFI_PORT_UNREACHABLE`: No service is listening on the remote port. `RxData` is set to `NULL` in this situation.
  - `EFI_ICMP_ERROR`: Some other Internet Control Message Protocol (ICMP) error report was received. For example, packets are being sent too fast for the destination to receive them and the destination sent an ICMP source quench report. `RxData` is set to `NULL` in this situation.
  - `EFI_DEVICE_ERROR`: An unexpected system or network error occurred.
  - `EFI_SECURITY_VIOLATION`: The transmit or receive was failed because of IPsec policy check.
- **RxData**: When this token is used for receiving, `RxData` is a pointer to `EFI_UDP6_RECEIVE_DATA`. Type `EFI_UDP6_RECEIVE_DATA` is defined below.
- **TxData**: When this token is used for transmitting, `TxData` is a pointer to `EFI_UDP6_TRANSMIT_DATA`. Type `EFI_UDP6_TRANSMIT_DATA` is defined below.
The **EFI_UDP6_COMPLETION_TOKEN** structures are used for both transmit and receive operations. When used for transmitting, the `Event` and `TxData` fields must be filled in by the EFI UDPv6 Protocol client. After the transmit operation completes, the `Status` field is updated by the EFI UDPv6 Protocol and the `Event` is signaled.

When used for receiving, only the `Event` field must be filled in by the EFI UDPv6 Protocol client. After a packet is received, `RxData` and `Status` are filled in by the EFI UDPv6 Protocol and the `Event` is signaled.

```c
//***************************************************************
// EFI_UDP6_RECEIVE_DATA
//***************************************************************
typedef struct {
    EFI_TIME           TimeStamp;
    EFI_EVENT          RecycleSignal;
    EFI_UDP6_SESSION_DATA   UdpSession;
    UINT32              DataLength;
    UINT32              FragmentCount;
    EFI_UDP6_FRAGMENT_DATA  FragmentTable[1];
} EFI_UDP6_RECEIVE_DATA;
```

- **TimeStamp**: Time when the EFI UDPv6 Protocol accepted the packet. `TimeStamp` is zero filled if timestamps are disabled or unsupported.
- **RecycleSignal**: Indicates the event to signal when the received data has been processed.
- **UdpSession**: The UDP session data including `SourceAddress`, `SourcePort`, `DestinationAddress`, and `DestinationPort`. Type `EFI_UDP6_SESSION_DATA` is defined below.
- **DataLength**: The sum of the fragment data length.
- **FragmentCount**: Number of fragments. Maybe zero.
- **FragmentTable**: Array of fragment descriptors. Maybe zero. Type `EFI_UDP6_FRAGMENT_DATA` is defined below.

**EFI_UDP6_RECEIVE_DATA** is filled by the EFI UDPv6 Protocol driver when this EFI UDPv6 Protocol instance receives an incoming packet. If there is a waiting token for incoming packets, the `CompletionToken.Packet.RxData` field is updated to this incoming packet and the `CompletionToken.Event` is signaled. The EFI UDPv6 Protocol client must signal the `RecycleSignal` after processing the packet.

**FragmentTable** could contain multiple buffers that are not in the continuous memory locations. The EFI UDPv6 Protocol client might need to combine two or more buffers in **FragmentTable** to form their own protocol header.
typedef struct {
    EFI_IPv6_ADDRESS SourceAddress;
    UINT16 SourcePort;
    EFI_IPv6_ADDRESS DestinationAddress;
    UINT16 DestinationPort;
} EFI_UDP6_SESSION_DATA;

SourceAddress Address from which this packet is sent. This field should not be used when sending packets.
SourcePort Port from which this packet is sent. It is in host byte order. This field should not be used when sending packets.
DestinationAddress Address to which this packet is sent. When sending packet, it’ll be ignored if it is zero.
DestinationPort Port to which this packet is sent. When sending packet, it’ll be ignored if it is zero.

The EFI_UDP6_SESSION_DATA is used to retrieve the settings when receiving packets or to override the existing settings (only DestinationAddress and DestinationPort can be overridden) of this EFI UDPv6 Protocol instance when sending packets.

typedef struct {
    UINT32 FragmentLength;
    VOID *FragmentBuffer;
} EFI_UDP6_FRAGMENT_DATA;

FragmentLength Length of the fragment data buffer.
FragmentBuffer Pointer to the fragment data buffer.

EFI_UDP6_FRAGMENT_DATA allows multiple receive or transmit buffers to be specified. The purpose of this structure is to avoid copying the same packet multiple times.
 typedef struct {
    EFI_UDP6_SESSION_DATA *UdpSessionData;
    UINT32 DataLength;
    UINT32 FragmentCount;
    EFI_UDP6_FRAGMENT_DATA FragmentTable[1];
} EFI_UDP6_TRANSMIT_DATA;

UdpSessionData is not NULL, the data that is used to override the transmitting settings. Only the two fields UdpSessionData.DestinationAddress and UdpSessionData.DestionPort can be used as the transmitting setting filed. Type EFI_UDP6_SESSION_DATA is defined above.

- **DataLength**: Sum of the fragment data length. Must not exceed the maximum UDP packet size.
- **FragmentCount**: Number of fragments.
- **FragmentTable**: Array of fragment descriptors. Type EFI_UDP6_FRAGMENT_DATA is defined above.

The EFI UDPv6 Protocol client must fill this data structure before sending a packet. The packet may contain multiple buffers that may not be in a continuous memory location.

**Status Codes Returned**

<table>
<thead>
<tr>
<th>Code</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>EFI_SUCCESS</td>
<td>The data has been queued for transmission.</td>
</tr>
<tr>
<td>EFI_NOT_STARTED</td>
<td>This EFI UDPv6 Protocol instance has not been started.</td>
</tr>
<tr>
<td>EFI_NO_MAPPING</td>
<td>The underlying IPv6 driver was responsible for choosing a source address for this instance, but no source address was available for use.</td>
</tr>
<tr>
<td>Code</td>
<td>Description</td>
</tr>
<tr>
<td>--------------------</td>
<td>---------------------------------------------------------------------------------------------------------------------------------------------</td>
</tr>
</tbody>
</table>
| EFI_INVALID_PARAMETER | One or more of the following are **TRUE**:  
This is **NULL**.  
Token is **NULL**.  
Token.Event is **NULL**.  
Token.Packet.TxData is **NULL**.  
Token.Packet.TxData.FragmentCount is zero.  
Token.Packet.TxData.DataLength is not equal to the sum of fragment lengths.  
One or more of the  
Token.Packet.TxData.FragmentTable[].FragmentLength fields is zero.  
One or more of the  
Token.Packet.TxData.FragmentTable[].FragmentBuffer fields is **NULL**.  
Token.Packet.TxData.UdpSessionData is **NULL** and this instance’s UdpConfigData. RemoteAddress is unspecified.  
  
Token.Packet.TxData.UdpSessionData.DestinationAddress is non-zero when DestinationAddress is configured as non-zero when doing **Configure()** for this EFI Udp6 protocol instance.  
Token.Packet.TxData.UdpSessionData.DestinationAddress is zero when DestinationAddress is unspecified when doing **Configure()** for this EFI Udp6 protocol instance.  

<table>
<thead>
<tr>
<th>EFI_ACCESS_DENIED</th>
<th>The transmit completion token with the same Token.Event was already in the transmit queue.</th>
</tr>
</thead>
<tbody>
<tr>
<td>EFI_NOT_READY</td>
<td>The completion token could not be queued because the transmit queue is full.</td>
</tr>
<tr>
<td>EFI_OUT_OF_RESOURCES</td>
<td>Could not queue the transmit data.</td>
</tr>
<tr>
<td>EFI_NOT_FOUND</td>
<td>There is no route to the destination network or address.</td>
</tr>
<tr>
<td>EFI_BAD_BUFFER_SIZE</td>
<td>The data length is greater than the maximum UDP packet size.</td>
</tr>
<tr>
<td>EFI_NO_MEDIA</td>
<td>There was a media error.</td>
</tr>
</tbody>
</table>
**EFI_UDP6_PROTOCOL.Receive()**

**Summary**
Places an asynchronous receive request into the receiving queue.

**Prototype**
```c
typedef EFI_STATUS (EFIAPI *EFI_UDP6_RECEIVE) (
    IN EFI_UDP6_PROTOCOL *This,
    IN EFI_UDP6_COMPLETION_TOKEN *Token
);
```

**Parameters**
- **This**: Pointer to the EFI_UDP6_PROTOCOL instance.
- **Token**: Pointer to a token that is associated with the receive data descriptor. Type EFI_UDP6_COMPLETION_TOKEN is defined in EFI_UDP6_PROTOCOL.Transmit().

**Description**
The `Receive()` function places a completion token into the receive packet queue. This function is always asynchronous.

The caller must fill in the **Token.Event** field in the completion token, and this field cannot be **NULL**. When the receive operation completes, the EFI UDPv6 Protocol driver updates the **Token.Status** and **Token.Packet.RxData** fields and the **Token.Event** is signaled. Providing a proper notification function and context for the event will enable the user to receive the notification and receiving status. That notification function is guaranteed to not be re-entered.

**Status Codes Returned**

<table>
<thead>
<tr>
<th>Code</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>EFI_SUCCESS</td>
<td>The receive completion token was cached.</td>
</tr>
<tr>
<td>EFI_NOT_STARTED</td>
<td>This EFI UDPv6 Protocol instance has not been started.</td>
</tr>
<tr>
<td>EFI_NO_MAPPING</td>
<td>The underlying IPv6 driver was responsible for choosing a source address for this instance, but no source address was available for use.</td>
</tr>
<tr>
<td>EFI_INVALID_PARAMETER</td>
<td>One or more of the following conditions is <strong>TRUE</strong>:</td>
</tr>
<tr>
<td></td>
<td>This is <strong>NULL</strong>. <strong>Token</strong> is <strong>NULL</strong>. <strong>Token.Event</strong> is <strong>NULL</strong>.</td>
</tr>
<tr>
<td>EFI_OUT_OF_RESOURCES</td>
<td>The receive completion token could not be queued due to a lack of system resources (usually memory).</td>
</tr>
<tr>
<td>EFI_DEVICE_ERROR</td>
<td>An unexpected system or network error occurred. The EFI UDPv6 Protocol instance has been reset to startup defaults.</td>
</tr>
<tr>
<td>EFI_ACCESS_DENIED</td>
<td>A receive completion token with the same <strong>Token.Event</strong> was already in the receive queue.</td>
</tr>
<tr>
<td>EFI_NOT_READY</td>
<td>The receive request could not be queued because the receive queue is full.</td>
</tr>
<tr>
<td>EFI_NO_MEDIA</td>
<td>There was a media error.</td>
</tr>
</tbody>
</table>
EFI_UDP6_PROTOCOL.Cancel()

Summary
Aborts an asynchronous transmit or receive request.

Prototype

```c
typedef EFI_STATUS
(EFIAPI *EFI_UDP6_CANCEL)(
    IN EFI_UDP6_PROTOCOL *This,
    IN EFI_UDP6_COMPLETION_TOKEN *Token
);
```

Parameters

- **This** Pointer to the EFI_UDP6_PROTOCOL instance.
- **Token** Pointer to a token that has been issued by EFI_UDP6_PROTOCOL.Transmit() or EFI_UDP6_PROTOCOL.Receive(). If NULL, all pending tokens are aborted. Type EFI_UDP6_COMPLETION_TOKEN is defined in EFI_UDP6_PROTOCOL.Transmit().

Description

The Cancel() function is used to abort a pending transmit or receive request. If the token is in the transmit or receive request queues, after calling this function, Token.Status will be set to EFI_ABORTED and then Token.Event will be signaled. If the token is not in one of the queues, which usually means that the asynchronous operation has completed, this function will not signal the token and EFI_NOT_FOUND is returned.

Status Codes Returned

<table>
<thead>
<tr>
<th>Status Code</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>EFI_SUCCESS</td>
<td>The asynchronous I/O request was aborted and Token.Event was signaled. When Token is NULL, all pending requests are aborted and their events are signaled.</td>
</tr>
<tr>
<td>EFI_INVALID_PARAMETER</td>
<td>This is NULL.</td>
</tr>
<tr>
<td>EFI_NOT_STARTED</td>
<td>This instance has not been started.</td>
</tr>
<tr>
<td>EFI_NOT_FOUND</td>
<td>When Token is not NULL, the asynchronous I/O request was not found in the transmit or receive queue. It has either completed or was not issued by Transmit() and Receive().</td>
</tr>
</tbody>
</table>

EFI_UDP6_PROTOCOL.Poll()

Summary
Polls for incoming data packets and processes outgoing data packets.
Prototype

```
typedef
    EFI_STATUS
    (EFIAPI *EFI_UDP6_POLL) (IN EFI_UDP6_PROTOCOL *This);
```

Parameters

- `This`: Pointer to the `EFI_UDP6_PROTOCOL` instance.

Description

The `Poll()` function can be used by network drivers and applications to increase the rate that data packets are moved between the communications device and the transmit and receive queues.

In some systems, the periodic timer event in the managed network driver may not poll the underlying communications device fast enough to transmit and/or receive all data packets without missing incoming packets or dropping outgoing packets. Drivers and applications that are experiencing packet loss should try calling the `Poll()` function more often.

Status Codes Returned

<table>
<thead>
<tr>
<th>Code</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>EFI_SUCCESS</td>
<td>Incoming or outgoing data was processed.</td>
</tr>
<tr>
<td>EFI_INVALID_PARAMETER</td>
<td>This is <code>NULL</code>.</td>
</tr>
<tr>
<td>EFI_DEVICE_ERROR</td>
<td>An unexpected system or network error occurred.</td>
</tr>
<tr>
<td>EFI_TIMEOUT</td>
<td>Data was dropped out of the transmit and/or receive queue. Consider increasing the polling rate.</td>
</tr>
</tbody>
</table>

30.3 EFI MTFTPv4 Protocol

The following sections defines the EFI MTFTPv4 Protocol interface that is built upon the EFI UDPv4 Protocol.

**EFI_MTFTP4_SERVICE_BINDING_PROTOCOL**

Summary

The EFI MTFTPv4 Service Binding Protocol is used to locate communication devices that are supported by an EFI MTFTPv4 Protocol driver and to create and destroy instances of the EFI MTFTPv4 Protocol child protocol driver that can use the underlying communications device.
GUID
#define EFI_MTFTP4_SERVICE_BINDING_PROTOCOL_GUID \ 
{0x2e800be,0x8f01,0x4aa6,\ 
{0x94,0x6b,0xd7,0x13,0x88,0xe1,0x83,0x3f}}

Description
A network application or driver that requires MTFTPv4 I/O services can use one of the protocol handler services, such as BS->LocateHandleBuffer(), to search for devices that publish an EFI MTFTPv4 Service Binding Protocol GUID. Each device with a published EFI MTFTPv4 Service Binding Protocol GUID supports the EFI MTFTPv4 Protocol service and may be available for use.

After a successful call to the EFI_MTFTP4_SERVICE_BINDING_PROTOCOL.CreateChild() function, the newly created child EFI MTFTPv4 Protocol driver instance is in an unconfigured state; it is not ready to transfer data.

Before a network application terminates execution, every successful call to the EFI_MTFTP4_SERVICE_BINDING_PROTOCOL.CreateChild() function must be matched with a call to the EFI_MTFTP4_SERVICE_BINDING_PROTOCOL.DestroyChild() function.

Each instance of the EFI MTFTPv4 Protocol driver can support one file transfer operation at a time. To download two files at the same time, two instances of the EFI MTFTPv4 Protocol driver will need to be created.

EFI_MTFTP4_PROTOCOL

Summary
The EFI MTFTPv4 Protocol provides basic services for client-side unicast and/or multicast TFTP operations.

GUID
#define EFI_MTFTP4_PROTOCOL_GUID \ 
{0x78247c57,0x63db,0x4708,\ 
{0x99,0xc2,0xa8,0xb4,0xa9,0xa6,0x1f,0x6b}}

Protocol Interface Structure
typedef struct _EFI_MTFTP4_PROTOCOL {
    EFI_MTFTP4_GET_MODE_DATA  GetModeData;
    EFI_MTFTP4_CONFIGURE     Configure;
    EFI_MTFTP4_GET_INFO      GetInfo;
    EFI_MTFTP4_PARSE_OPTIONS ParseOptions;
    EFI_MTFTP4_READ_FILE     ReadFile;
    EFI_MTFTP4_WRITE_FILE    WriteFile;
    EFI_MTFTP4_READ_DIRECTORY ReadDirectory;
    EFI_MTFTP4_POLL          Poll;
} EFI_MTFTP4_PROTOCOL;

Parameters
UEFI Specification, Version 2.9

Network Protocols — UDP and MTFTP

GetModeData
Reads the current operational settings. See the GetModeData() function description.

Configure
Initializes, changes, or resets the operational settings for this instance of the EFI MTFTPv4 Protocol driver. See the Configure() function description.

GetInfo
Retrieves information about a file from an MTFTPv4 server. See the GetInfo() function description.

ParseOptions
Parses the options in an MTFTPv4 OACK (options acknowledgement) packet. See the ParseOptions() function description.

ReadFile
Downloads a file from an MTFTPv4 server. See the ReadFile() function description.

WriteFile
Uploads a file to an MTFTPv4 server. This function may be unsupported in some EFI implementations. See the WriteFile() function description.

ReadDirectory
Downloads a related file “directory” from an MTFTPv4 server. This function may be unsupported in some EFI implementations. See the ReadDirectory() function description.

Poll
Polls for incoming data packets and processes outgoing data packets. See the Poll() function description.

Description
The EFI_MTFTP4_PROTOCOL is designed to be used by UEFI drivers and applications to transmit and receive data files. The EFI MTFTPv4 Protocol driver uses the underlying EFI UDPv4 Protocol driver and EFI IPv4 Protocol driver.

EFI_MTFTP4_PROTOCOL.GetModeData()

Summary
Reads the current operational settings.

Prototype
typedef
    EFI_STATUS
    (EFIAPI *EFI_MTFTP4_GET_MODE_DATA)(
    IN EFI_MTFTP4_PROTOCOL *This,
    OUT EFI_MTFTP4_MODE_DATA *ModeData
    );

Parameters
This
    Pointer to the EFI_MTFTP4_PROTOCOL instance.

ModeData
    Pointer to storage for the EFI MTFTPv4 Protocol driver mode data. Type EFI_MTFTP4_MODE_DATA is defined in “Related Definitions” below.
Description
The GetModeData() function reads the current operational settings of this EFI MTFTPv4 Protocol driver instance.

Related Definitions

```c
typedef struct {
  EFI_MTFTP4_CONFIG_DATA   ConfigData;
  UINT8                     SupportedOptionCount;
  **SupportedOptions;
  UINT8                     UnsupportedOptionCount;
  **UnsupportedOptions;
} EFI_MTFTP4_MODE_DATA;
```

- **ConfigData**: The configuration data of this instance. Type EFI_MTFTP4_CONFIG_DATA is defined below.
- **SupportedOptionCount**: The number of option strings in the following SupportedOptions array.
- **SupportedOptions**: An array of pointers to null-terminated ASCII option strings that are recognized and supported by this EFI MTFTPv4 Protocol driver implementation.
- **UnsupportedOptionCount**: An array of pointers to null-terminated ASCII option strings that are recognized but not supported by this EFI MTFTPv4 Protocol driver implementation.
- **UnsupportedOptions**: An array of option strings that are recognized but are not supported by this EFI MTFTPv4 Protocol driver implementation.

The EFI_MTFTP4_MODE_DATA structure describes the operational state of this instance.
typedef struct {
  BOOLEAN UseDefaultSetting;
  EFI_IPv4_ADDRESS StationIp;
  EFI_IPv4_ADDRESS SubnetMask;
  UINT16 LocalPort;
  EFI_IPv4_ADDRESS GatewayIp;
  EFI_IPv4_ADDRESS ServerIp;
  UINT16 InitialServerPort;
  UINT16 TryCount;
  UINT16 TimeoutValue;
} EFI_MTFTP4_CONFIG_DATA;

UseDefaultSetting  Set to TRUE to use the default station address/subnet mask and the default route table information.
StationIp  If UseDefaultSetting is FALSE, indicates the station address to use.
SubnetMask  If UseDefaultSetting is FALSE, indicates the subnet mask to use.
LocalPort  Local port number. Set to zero to use the automatically assigned port number.
GatewayIp  if UseDefaultSetting is FALSE, indicates the gateway IP address to use.
ServerIp  The IP address of the MTFTPv4 server.
InitialServerPort  The initial MTFTPv4 server port number. Request packets are sent to this port. This number is almost always 69 and using zero defaults to 69.
TryCount  The number of times to transmit MTFTPv4 request packets and wait for a response.
TimeoutValue  The number of seconds to wait for a response after sending the MTFTPv4 request packet.

The EFI_MTFTP4_CONFIG_DATA structure is used to report and change MTFTPv4 session parameters.
Status Codes Returned

<table>
<thead>
<tr>
<th>Code</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>EFI_SUCCESS</td>
<td>The configuration data was successfully returned.</td>
</tr>
<tr>
<td>EFI_OUT_OF_RESOURCES</td>
<td>The required mode data could not be allocated.</td>
</tr>
<tr>
<td>EFI_INVALID_PARAMETER</td>
<td>This is NULL or ModeData is NULL</td>
</tr>
</tbody>
</table>

**EFI_MTFTP4_PROTOCOL.Configure()**

**Summary**
Initializes, changes, or resets the default operational setting for this EFI MTFTPv4 Protocol driver instance.

**Prototype**

```c
typedef
EFI_STATUS
(EIFIAPI *EFI_MTFTP4_CONFIGURE)(
    IN EFI_MTFTP4_PROTOCOL *This,
    IN EFI_MTFTP4_CONFIG_DATA *MftpConfigData OPTIONAL
);
```

**Parameters**
- *This* Pointer to the EFI_MTFTP4_PROTOCOL instance.
- *MftpConfigData* Pointer to the configuration data structure. Type EFI_MTFTP4_CONFIG_DATA is defined in EFI_MTFTP4_PROTOCOL.GetModeData().

**Description**
The `Configure()` function is used to set and change the configuration data for this EFI MTFTPv4 Protocol driver instance. The configuration data can be reset to startup defaults by calling `Configure()` with *MftpConfigData* set to NULL. Whenever the instance is reset, any pending operation is aborted. By changing the EFI MTFTPv4 Protocol driver instance configuration data, the client can connect to different MTFTPv4 servers. The configuration parameters in *MftpConfigData* are used as the default parameters in later MTFTPv4 operations and can be overridden in later operations.
Status Codes Returned

<table>
<thead>
<tr>
<th>Code</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>EFI_SUCCESS</td>
<td>The EFI MTFTPv4 Protocol driver was configured successfully.</td>
</tr>
<tr>
<td>EFI_INVALID_PARAMETER</td>
<td>One or more following conditions are <strong>TRUE</strong>:</td>
</tr>
<tr>
<td></td>
<td>• <strong>This</strong> is <strong>NULL</strong>.</td>
</tr>
<tr>
<td></td>
<td>• <strong>MtftpConfigData.UseDefaultSetting</strong> is <strong>FALSE</strong> and</td>
</tr>
<tr>
<td></td>
<td>• <strong>MtftpConfigData.StationIp</strong> is not a valid IPv4 unicast address.</td>
</tr>
<tr>
<td></td>
<td>• <strong>MtftpConfigData.UseDefaultSetting</strong> is <strong>FALSE</strong> and</td>
</tr>
<tr>
<td></td>
<td>• <strong>MtftpConfigData.SubnetMask</strong> is invalid.</td>
</tr>
<tr>
<td></td>
<td>• <strong>MtftpConfigData.ServerIp</strong> is not a valid IPv4 unicast address.</td>
</tr>
<tr>
<td></td>
<td>• <strong>MtftpConfigData.UseDefaultSetting</strong> is <strong>FALSE</strong> and</td>
</tr>
<tr>
<td></td>
<td>• <strong>MtftpConfigData.GatewayIp</strong> is not a valid IPv4 unicast address or is not</td>
</tr>
<tr>
<td></td>
<td>in the same subnet with station address.</td>
</tr>
<tr>
<td>EFI_ACCESS_DENIED</td>
<td>The EFI configuration could not be changed at this time because there is</td>
</tr>
<tr>
<td></td>
<td>one MTFTP background operation in progress.</td>
</tr>
<tr>
<td>EFI_NO_MAPPING</td>
<td>When using a default address, configuration (DHCP, BOOTP, RARP, etc.)</td>
</tr>
<tr>
<td></td>
<td>has not finished yet.</td>
</tr>
<tr>
<td>EFI_UNSUPPORTED</td>
<td>A configuration protocol (DHCP, BOOTP, RARP, etc.) could not be located</td>
</tr>
<tr>
<td></td>
<td>when clients choose to use the default address settings.</td>
</tr>
<tr>
<td>EFI_OUT_OF_RESOURCES</td>
<td>The EFI MTFTPv4 Protocol driver instance data could not be allocated.</td>
</tr>
<tr>
<td>EFI_DEVICE_ERROR</td>
<td>An unexpected system or network error occurred. The EFI MTFTPv4 Protocol</td>
</tr>
<tr>
<td></td>
<td>driver instance is not configured.</td>
</tr>
</tbody>
</table>

**EFI_MTFTP4_PROTOCOL.GetInfo()**

**Summary**

Gets information about a file from an MTFTPv4 server.

**Prototype**

```c
typedef
EFI_STATUS
(EIFIAPI *EFI_MTFTP4_GET_INFO)(
    IN EFI_MTFTP4_PROTOCOL *This,
    IN EFI_MTFTP4_OVERRIDE_DATA *OverrideData OPTIONAL,
    IN UINT8 *Filename,
    IN UINT8 *ModeStr OPTIONAL,
    IN UINT8 OptionCount,
    IN EFI_MTFTP4_OPTION *OptionList OPTIONAL,
    OUT UINT32 *PacketLength,
    OUT EFI_MTFTP4_PACKET **Packet OPTIONAL
    )
```

**Parameters**

- **This** Pointer to the **EFI_MTFTP4_PROTOCOL** instance.
OverrideData

Data that is used to override the existing parameters. If NULL, the default parameters that were set in the
\texttt{EFI\_MTFTP4\_PROTOCOL\_Configure()} function are used. Type \texttt{EFI\_MTFTP4\_OVERRIDE\_DATA} is defined in “Related
Definitions” below.

Filename

Pointer to a null-terminated ASCII file name string.

ModeStr

Pointer to a null-terminated ASCII mode string. If NULL, “octet”
will be used.

OptionCount

Number of option/value string pairs in \texttt{OptionList}.

OptionList

Pointer to array of option/value string pairs. Ignored if
\texttt{OptionCount} is zero. Type \texttt{EFI\_MTFTP4\_OPTION} is defined in
“Related Definitions” below.

PacketLength

The number of bytes in the returned packet.

Packet

The pointer to the received packet. This buffer must be freed
by the caller. Type \texttt{EFI\_MTFTP4\_PACKET} is defined in “Related
Definitions” below.

Description

The \texttt{GetInfo()} function assembles an MTFTPv4 request packet with options; sends it to the MTFTPv4
server; and may return an MTFTPv4 OACK, MTFTPv4 ERROR, or ICMP ERROR packet. Retries occur only if
no response packets are received from the MTFTPv4 server before the timeout expires.

Related Definitions

```c
//**************************************************************
// EFI_MTFTP_OVERRIDE_DATA
//**************************************************************
typedef struct {
  EFI_IPv4_ADDRESS GatewayIp;
  EFI_IPv4_ADDRESS ServerIp;
  UINT16 ServerPort;
  UINT16 TryCount;
  UINT16 TimeoutValue;
} EFI_MTFTP4_OVERRIDE_DATA;
```

GatewayIp

IP address of the gateway. If set to 0.0.0.0, the default gateway
address that was set by the
\texttt{EFI\_MTFTP4\_PROTOCOL\_Configure()} function will not be
 overridden.

ServerIp

IP address of the MTFTPv4 server. If set to 0.0.0.0, it will use
the value that was set by the
\texttt{EFI\_MTFTP4\_PROTOCOL\_Configure()} function.

ServerPort

MTFTPv4 server port number. If set to zero, it will use the
value that was set by the
\texttt{EFI\_MTFTP4\_PROTOCOL\_Configure()} function.
TryCount
Number of times to transmit MTFTPv4 request packets and wait for a response. If set to zero, it will use the value that was set by the `EFI_MTFTP4_PROTOCOL.Configure()` function.

TimeoutValue
Number of seconds to wait for a response after sending the MTFTPv4 request packet. If set to zero, it will use the value that was set by the `EFI_MTFTP4_PROTOCOL.Configure()` function.

The `EFI_MTFTP4_OVERRIDE_DATA` structure is used to override the existing parameters that were set by the `EFI_MTFTP4_PROTOCOL.Configure()` function.

```c
//***************************************************************
// EFI_MTFTP4_OPTION
//***************************************************************
typedef struct {
    UINT8    *OptionStr;
    UINT8    *ValueStr;
} EFI_MTFTP4_OPTION;
```

OptionStr Pointer to the null-terminated ASCII MTFTPv4 option string.
ValueStr Pointer to the null-terminated ASCII MTFTPv4 value string.

```c
#pragma pack(1)
//***************************************************************
// EFI_MTFTP4_PACKET
//***************************************************************
typedef union {
    UINT16       OpCode;
    EFI_MTFTP4_REQ_HEADER    Rrq, Wrq;
    EFI_MTFTP4_OACK_HEADER   Oack;
    EFI_MTFTP4_DATA_HEADER   Data;
    EFI_MTFTP4_ACK_HEADER    Ack;
    // This field should be ignored and treated as reserved
    EFI_MTFTP4_DATA8_HEADER  Data8;
    // This field should be ignored and treated as reserved
    EFI_MTFTP4_ACK8_HEADER   Ack8;
    EFI_MTFTP4_ERROR_HEADER  Error;
} EFI_MTFTP4_PACKET;
```
typedef struct {
    UINT16 OpCode;
    UINT8 Filename[1];
} EFI_MTFTP4_REQ_HEADER;

typedef struct {
    UINT16 OpCode;
    UINT8 Data[1];
} EFI_MTFTP4_OACK_HEADER;

typedef struct {
    UINT16 OpCode;
    UINT16 Block;
    UINT8 Data[1];
} EFI_MTFTP4_DATA_HEADER;

typedef struct {
    UINT16 OpCode;
    UINT16 Block[1];
} EFI_MTFTP4_ACK_HEADER;

typedef struct {
    UINT16 OpCode;
    UINT64 Block;
    UINT8 Data[1];
} EFI_MTFTP4_DATA8_HEADER;

typedef struct {
    UINT16 OpCode;
    UINT64 Block;
    UINT8 Data[1];
} EFI_MTFTP4_ACK8_HEADER;
typedef struct {
  UINT16    OpCode;
  UINT64    Block[1];
} EFI_MTFTP4_ACK8_HEADER;

typedef struct {
  UINT16    OpCode;
  UINT16    ErrorCode;
  UINT8     ErrorMessage[1];
} EFI_MTFTP4_ERROR_HEADER;

#pragma pack()
Table 30-1 Descriptions of Parameters in MTFTPv4 Packet Structures

<table>
<thead>
<tr>
<th>Data Structure</th>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>EFI_MTFTP4_PACKET</strong></td>
<td>OpCode</td>
<td>Type of packets as defined by the MTFTPv4 packet opcodes. Opcode values are defined below.</td>
</tr>
<tr>
<td></td>
<td>Rrq, Wrq</td>
<td>Read request or write request packet header. See the description for <strong>EFI_MTFTP4_REQ_HEADER</strong> below in this table.</td>
</tr>
<tr>
<td></td>
<td>Oack</td>
<td>Option acknowledge packet header. See the description for <strong>EFI_MTFTP4_OACK_HEADER</strong> below in this table.</td>
</tr>
<tr>
<td></td>
<td>Data</td>
<td>Data packet header. See the description for <strong>EFI_MTFTP4_DATA_HEADER</strong> below in this table.</td>
</tr>
<tr>
<td></td>
<td>Ack</td>
<td>Acknowledgement packet header. See the description for <strong>EFI_MTFTP4_ACK_HEADER</strong> below in this table.</td>
</tr>
<tr>
<td></td>
<td>Data8</td>
<td>This field should be ignored and treated as reserved. Data packet header with big block number. See the description for <strong>EFI_MTFTP4_DATA8_HEADER</strong> below in this table.</td>
</tr>
<tr>
<td></td>
<td>Ack8</td>
<td>This field should be ignored and treated as reserved. Acknowledgement header with big block number. See the description for <strong>EFI_MTFTP4_ACK8_HEADER</strong> below in this table.</td>
</tr>
<tr>
<td></td>
<td>Error</td>
<td>Error packet header. See the description for <strong>EFI_MTFTP4_ERROR_HEADER</strong> below in this table.</td>
</tr>
<tr>
<td><strong>EFI_MTFTP4_REQ_HEADER</strong></td>
<td>OpCode</td>
<td>For this packet type, OpCode = <strong>EFI_MTFTP4_OPCODE_RRQ</strong> for a read request or OpCode = <strong>EFI_MTFTP4_OPCODE_WRQ</strong> for a write request.</td>
</tr>
<tr>
<td></td>
<td>Filename</td>
<td>The file name to be downloaded or uploaded.</td>
</tr>
<tr>
<td><strong>EFI_MTFTP4_OACK_HEADER</strong></td>
<td>OpCode</td>
<td>For this packet type, OpCode = <strong>EFI_MTFTP4_OPCODE_OACK</strong></td>
</tr>
<tr>
<td></td>
<td>Data</td>
<td>The option strings in the option acknowledgement packet.</td>
</tr>
<tr>
<td><strong>EFI_MTFTP4_DATA_HEADER</strong></td>
<td>OpCode</td>
<td>For this packet type, OpCode = <strong>EFI_MTFTP4_OPCODE_DATA</strong></td>
</tr>
<tr>
<td></td>
<td>Block</td>
<td>Block number of this data packet.</td>
</tr>
<tr>
<td></td>
<td>Data</td>
<td>The content of this data packet.</td>
</tr>
<tr>
<td>Data Structure</td>
<td>Parameter</td>
<td>Description</td>
</tr>
<tr>
<td>-------------------------</td>
<td>---------------</td>
<td>----------------------------------------------------------------------------</td>
</tr>
<tr>
<td>EFI_MTFTP4_ACK_HEADER</td>
<td>OpCode</td>
<td>For this packet type, OpCode = EFI_MTFTP4_OPCODE_ACK.</td>
</tr>
<tr>
<td></td>
<td>Block</td>
<td>The block number of the data packet that is being acknowledged.</td>
</tr>
<tr>
<td>EFI_MTFTP4_DATA8_HEADER</td>
<td>OpCode</td>
<td>This field should be ignored and treated as reserved.</td>
</tr>
<tr>
<td></td>
<td>Block</td>
<td>This field should be ignored and treated as reserved.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>The block number of data packet.</td>
</tr>
<tr>
<td></td>
<td>Data</td>
<td>This field should be ignored and treated as reserved.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>The content of this data packet.</td>
</tr>
<tr>
<td>EFI_MTFTP4_ACK8_HEADER</td>
<td>OpCode</td>
<td>This field should be ignored and treated as reserved.</td>
</tr>
<tr>
<td></td>
<td>Block</td>
<td>This field should be ignored and treated as reserved.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>The block number of the data packet that is being acknowledged.</td>
</tr>
<tr>
<td>EFI_MTFTP4_ERROR_HEADER</td>
<td>OpCode</td>
<td>For this packet type, OpCode = EFI_MTFTP4_OPCODE_ERROR.</td>
</tr>
<tr>
<td></td>
<td>ErrorCode</td>
<td>The error number as defined by the MTFTPv4 packet error codes. Values for ErrorCode are defined below.</td>
</tr>
<tr>
<td></td>
<td>ErrorMessage</td>
<td>Error message string.</td>
</tr>
</tbody>
</table>
Following is a description of the fields in the above definition.

<table>
<thead>
<tr>
<th>Field</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>EFI_MTFTP4_OPCODE_RRQ</td>
<td>The MTFTPv4 packet is a read request.</td>
</tr>
<tr>
<td>EFI_MTFTP4_OPCODE_WRQ</td>
<td>The MTFTPv4 packet is a write request.</td>
</tr>
<tr>
<td>EFI_MTFTP4_OPCODE_DATA</td>
<td>The MTFTPv4 packet is a data packet.</td>
</tr>
<tr>
<td>EFI_MTFTP4_OPCODE_ACK</td>
<td>The MTFTPv4 packet is an acknowledgement packet.</td>
</tr>
<tr>
<td>EFI_MTFTP4_OPCODE_ERROR</td>
<td>The MTFTPv4 packet is an error packet.</td>
</tr>
<tr>
<td>EFI_MTFTP4_OPCODE_OACK</td>
<td>The MTFTPv4 packet is an option acknowledgement packet.</td>
</tr>
<tr>
<td>EFI_MTFTP4_OPCODE_DIR</td>
<td>The MTFTPv4 packet is a directory query packet.</td>
</tr>
<tr>
<td>EFI_MTFTP4_OPCODE_DATA8</td>
<td>This field should be ignored and treated as reserved. The MTFTPv4 packet is a data packet with a big block number.</td>
</tr>
<tr>
<td>EFI_MTFTP4_OPCODE_ACK8</td>
<td>This field should be ignored and treated as reserved. The MTFTPv4 packet is an acknowledgement packet with a big block number.</td>
</tr>
</tbody>
</table>
// MTFTP ERROR Packet ErrorCodes

#define EFI_MTFTP4_ERRORCODE_NOT_DEFINED           0
#define EFI_MTFTP4_ERRORCODE_FILE_NOT_FOUND        1
#define EFI_MTFTP4_ERRORCODE_ACCESS_VIOLATION      2
#define EFI_MTFTP4_ERRORCODE_DISK_FULL             3
#define EFI_MTFTP4_ERRORCODE_ILLEGAL_OPERATION     4
#define EFI_MTFTP4_ERRORCODE_UNKNOWN_TRANSFER_ID   5
#define EFI_MTFTP4_ERRORCODE_FILE_ALREADY_EXISTS   6
#define EFI_MTFTP4_ERRORCODE_NO_SUCH_USER          7
#define EFI_MTFTP4_ERRORCODE_REQUEST_DENIED        8

<table>
<thead>
<tr>
<th>Error Code</th>
<th>Message</th>
</tr>
</thead>
<tbody>
<tr>
<td>EFI_MTFTP4_ERRORCODE_NOT_DEFINED</td>
<td>The error code is not defined. See the error message in the packet (if any) for details.</td>
</tr>
<tr>
<td>EFI_MTFTP4_ERRORCODE_FILE_NOT_FOUND</td>
<td>The file was not found.</td>
</tr>
<tr>
<td>EFI_MTFTP4_ERRORCODE_ACCESS_VIOLATION</td>
<td>There was an access violation.</td>
</tr>
<tr>
<td>EFI_MTFTP4_ERRORCODE_DISK_FULL</td>
<td>The disk was full or its allocation was exceeded.</td>
</tr>
<tr>
<td>EFI_MTFTP4_ERRORCODE_ILLEGAL_OPERATION</td>
<td>The MTFTPv4 operation was illegal.</td>
</tr>
<tr>
<td>EFI_MTFTP4_ERRORCODE_UNKNOWN_TRANSFER_ID</td>
<td>The transfer ID is unknown.</td>
</tr>
<tr>
<td>EFI_MTFTP4_ERRORCODE_FILE_ALREADY_EXISTS</td>
<td>The file already exists.</td>
</tr>
<tr>
<td>EFI_MTFTP4_ERRORCODE_NO_SUCH_USER</td>
<td>There is no such user.</td>
</tr>
<tr>
<td>EFI_MTFTP4_ERRORCODE_REQUEST_DENIED</td>
<td>The request has been denied due to option negotiation.</td>
</tr>
</tbody>
</table>

Status Codes Returned

<table>
<thead>
<tr>
<th>Code</th>
<th>Message</th>
</tr>
</thead>
<tbody>
<tr>
<td>EFI_SUCCESS</td>
<td>An MTFTPv4 OACK packet was received and is in the <strong>Packet</strong>.</td>
</tr>
<tr>
<td>EFI_INVALID_PARAMETER</td>
<td>One or more of the following conditions is <strong>TRUE:</strong></td>
</tr>
<tr>
<td></td>
<td>• This is <strong>NULL</strong></td>
</tr>
<tr>
<td></td>
<td>• Filename is <strong>NULL</strong></td>
</tr>
<tr>
<td></td>
<td>• OptionCount is not zero and OptionList is <strong>NULL</strong></td>
</tr>
<tr>
<td></td>
<td>• One or more options in OptionList have wrong format.</td>
</tr>
<tr>
<td></td>
<td>• PacketLength is <strong>NULL</strong></td>
</tr>
<tr>
<td></td>
<td>• One or more IPv4 addresses in OverrideData are not valid unicast IPv4 addresses if OverrideData is not <strong>NULL</strong> and the addresses are not set to all zero.</td>
</tr>
<tr>
<td>EFI_UNSUPPORTED</td>
<td>One or more options in the OptionList are in the unsupported list of structure <strong>EFI_MTFTP4_MODE_DATA</strong>.</td>
</tr>
</tbody>
</table>


**EFI_MTFTP4_PROTOCOL.ParseOptions()**

**Summary**
Parses the options in an MTFTPv4 OACK packet.

**Prototype**

```c
typedef
EFI_STATUS
(EIFIAPI *EFI_MTFTP4_PARSE_OPTIONS)(
    IN EFI_MTFTP4_PROTOCOL *This,
    IN UINT32 PacketLen,
    IN EFI_MTFTP4_PACKET *Packet,
    OUT UINT32 *OptionCount,
    OUT EFI_MTFTP4_OPTION **OptionList OPTIONAL
);
```

**Parameters**

- **This**
  Pointer to the **EFI_MTFTP4_PROTOCOL** instance.

- **PacketLen**
  Length of the OACK packet to be parsed.

- **Packet**
  Pointer to the OACK packet to be parsed. Type **EFI_MTFTP4_PACKET** is defined in **EFI_MTFTP4_PROTOCOL.GetInfo()**.

---

<table>
<thead>
<tr>
<th>Code</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>EFI_NOT_STARTED</td>
<td>The EFI MTFTPv4 Protocol driver has not been started.</td>
</tr>
<tr>
<td>EFI_NO_MAPPING</td>
<td>When using a default address, configuration (DHCP, BOOTP, RARP, etc.) has</td>
</tr>
<tr>
<td></td>
<td>not finished yet.</td>
</tr>
<tr>
<td>EFI_ACCESS_DENIED</td>
<td>The previous operation has not completed yet.</td>
</tr>
<tr>
<td>EFI_OUT_OF_RESOURCES</td>
<td>Required system resources could not be allocated.</td>
</tr>
<tr>
<td>EFI_TFTP_ERROR</td>
<td>An MTFTPv4 ERROR packet was received and is in the <strong>Packet</strong>.</td>
</tr>
<tr>
<td>EFI_NETWORK_UNREACHABLE</td>
<td>An ICMP network unreachable error packet was received and the <strong>Packet</strong> is set to <strong>NULL</strong>.</td>
</tr>
<tr>
<td>EFI_HOST_UNREACHABLE</td>
<td>An ICMP host unreachable error packet was received and the <strong>Packet</strong> is set to <strong>NULL</strong>.</td>
</tr>
<tr>
<td>EFI_PROTOCOL_UNREACHABLE</td>
<td>An ICMP protocol unreachable error packet was received and the <strong>Packet</strong> is set to <strong>NULL</strong>.</td>
</tr>
<tr>
<td>EFI_PORT_UNREACHABLE</td>
<td>An ICMP port unreachable error packet was received and the <strong>Packet</strong> is set to <strong>NULL</strong>.</td>
</tr>
<tr>
<td>EFI_ICMP_ERROR</td>
<td>Some other ICMP ERROR packet was received and the <strong>Packet</strong> is set to NULL.</td>
</tr>
<tr>
<td>EFI_PROTOCOL_ERROR</td>
<td>An unexpected MTFTPv4 packet was received and is in the <strong>Packet</strong>.</td>
</tr>
<tr>
<td>EFI_TIMEOUT</td>
<td>No responses were received from the MTFTPv4 server.</td>
</tr>
<tr>
<td>EFI_DEVICE_ERROR</td>
<td>An unexpected network error or system error occurred.</td>
</tr>
<tr>
<td>EFI_NO_MEDIA</td>
<td>There was a media error.</td>
</tr>
</tbody>
</table>
OptionCount
OptionList

Pointer to the number of options in following OptionList.
Pointer to EFI_MTFTP4_OPTION storage. Call the EFI Boot Service FreePool() to release the OptionList if the options in this OptionList are not needed any more. Type EFI_MTFTP4_OPTION is defined in EFI_MTFTP4_PROTOCOL.GetInfo().

Description
The ParseOptions() function parses the option fields in an MTFTPv4 OACK packet and returns the number of options that were found and optionally a list of pointers to the options in the packet.

If one or more of the option fields are not valid, then EFI_PROTOCOL_ERROR is returned and *OptionCount and *OptionList stop at the last valid option.

Status Codes Returned

<table>
<thead>
<tr>
<th>Code</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>EFI_SUCCESS</td>
<td>The OACK packet was valid and the OptionCount and OptionList parameters have been updated.</td>
</tr>
<tr>
<td>EFI_INVALID_PARAMETER</td>
<td>One or more of the following conditions is TRUE:</td>
</tr>
<tr>
<td></td>
<td>• PacketLen is 0.</td>
</tr>
<tr>
<td></td>
<td>• Packet is NULL or Packet is not a valid MTFTPv4 packet.</td>
</tr>
<tr>
<td></td>
<td>• OptionCount is NULL</td>
</tr>
<tr>
<td>EFI_NOT_FOUND</td>
<td>No options were found in the OACK packet.</td>
</tr>
<tr>
<td>EFI_OUT_OF_RESOURCES</td>
<td>Storage for the OptionList array cannot be allocated.</td>
</tr>
<tr>
<td>EFI_PROTOCOL_ERROR</td>
<td>One or more of the option fields is invalid.</td>
</tr>
</tbody>
</table>

EFI_MTFTP4_PROTOCOL.ReadFile()

Summary
Downloads a file from an MTFTPv4 server.

Prototype
typedef

EFI_STATUS
(EFIAPI *EFI_MTFTP4_READ_FILE)(
    IN EFI_MTFTP4_PROTOCOL   *This,
    IN EFI_MTFTP4_TOKEN      *Token
);

Parameters
This
Token

Pointer to the EFI_MTFTP4_PROTOCOL instance.
Pointer to the token structure to provide the parameters that are used in this operation. Type EFI_MTFTP4_TOKEN is defined in “Related Definitions” below.
Description

The **ReadFile()** function is used to initialize and start an MTFTPv4 download process and optionally wait for completion. When the download operation completes, whether successfully or not, the **Token.Status** field is updated by the EFI MTFTPv4 Protocol driver and then **Token.Event** is signaled (if it is not **NULL**).

Data can be downloaded from the MTFTPv4 server into either of the following locations:

- A fixed buffer that is pointed to by **Token.Buffer**
- A download service function that is pointed to by **Token.CheckPacket**

If both **Token.Buffer** and **Token.CheckPacket** are used, then **Token.CheckPacket** will be called first. If the call is successful, the packet will be stored in **Token.Buffer**.

Related Definitions

```c
typedef struct {
    EFI_STATUS Status;
    EFI_EVENT Event;
    EFI_MTFTP4_OVERRIDE_DATA *OverrideData;
    UINT8 *Filename;
    UINT8 *ModeStr;
    UINT32 OptionCount;
    EFI_MTFTP4_OPTION *OptionList;
    UINT64 BufferSize;
    VOID *Buffer;
    VOID *Context;
    EFI_MTFTP4_CHECK_PACKET CheckPacket;
    EFI_MTFTP4_TIMEOUT_CALLBACK TimeoutCallback;
    EFI_MTFTP4_PACKET_NEEDED PacketNeeded;
} EFI_MTFTP4_TOKEN;
```

**Status**

The status that is returned to the caller at the end of the operation to indicate whether this operation completed successfully. Defined **Status** values are listed below.

**Event**

The event that will be signaled when the operation completes. If set to **NULL**, the corresponding function will wait until the read or write operation finishes. The type of **Event** must be **EVT_NOTIFY_SIGNAL**. The Task Priority Level (TPL) of **Event** must be lower than or equal to **TPL_CALLBACK**.

**OverrideData**

If not **NULL**, the data that will be used to override the existing configure data. Type **EFI_MTFTP4_OVERRIDE_DATA** is defined in **EFI_MTFTP4_PROTOCOL.GetInfo()**.

**Filename**

Pointer to the null-terminated ASCII file name string.
<table>
<thead>
<tr>
<th>Field</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>ModeStr</td>
<td>Pointer to the null-terminated ASCII mode string. If <strong>NULL</strong>, “octet” is used.</td>
</tr>
<tr>
<td>OptionCount</td>
<td>Number of option/value string pairs.</td>
</tr>
<tr>
<td>OptionList</td>
<td>Pointer to an array of option/value string pairs. Ignored if <strong>OptionCount</strong> is zero. Both a remote server and this driver implementation should support these options. If one or more options are unrecognized by this implementation, it is sent to the remote server without being changed. Type <strong>EFI_MTFTP4_OPTION</strong> is defined in <strong>EFI_MTFTP4_PROTOCOL.GetInfo()</strong>.</td>
</tr>
<tr>
<td>BufferSize</td>
<td>On input, the size, in bytes, of <strong>Buffer</strong>. On output, the number of bytes transferred.</td>
</tr>
<tr>
<td>Buffer</td>
<td>Pointer to the data buffer. Data that is downloaded from the MTFTPv4 server is stored here. Data that is uploaded to the MTFTPv4 server is read from here. Ignored if <strong>BufferSize</strong> is zero.</td>
</tr>
<tr>
<td>Context</td>
<td>Pointer to the context that will be used by <strong>CheckPacket</strong>, <strong>TimeoutCallback</strong> and <strong>PacketNeeded</strong>.</td>
</tr>
<tr>
<td>CheckPacket</td>
<td>Pointer to the callback function to check the contents of the received packet. Type <strong>EFI_MTFTP4_CHECK_PACKET</strong> is defined below.</td>
</tr>
<tr>
<td>TimeoutCallback</td>
<td>Pointer to the function to be called when a timeout occurs. Type <strong>EFI_MTFTP4_TIMEOUT_CALLBACK</strong> is defined below.</td>
</tr>
<tr>
<td>PacketNeeded</td>
<td>Pointer to the function to provide the needed packet contents. Only used in <strong>WriteFile()</strong> operation. Type <strong>EFI_MTFTP4_PACKET_NEEDED</strong> is defined below.</td>
</tr>
</tbody>
</table>

The **EFI_MTFTP4_TOKEN** structure is used for both the MTFTPv4 reading and writing operations. The caller uses this structure to pass parameters and indicate the operation context. After the reading or writing operation completes, the EFI MTFTPv4 Protocol driver updates the **Status** parameter and the **Event** is signaled if it is not **NULL**. The following table lists the status codes that are returned in the **Status** parameter.
Status Codes Returned in the Status Parameter

<table>
<thead>
<tr>
<th>Code</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>EFI_SUCCESS</td>
<td>The data file has been transferred successfully.</td>
</tr>
<tr>
<td>EFI_OUT_OF_RESOURCES</td>
<td>Required system resources could not be allocated.</td>
</tr>
<tr>
<td>EFIBUFFER_TOO_SMALL</td>
<td>BufferSize is not large enough to hold the downloaded data in downloading process.</td>
</tr>
<tr>
<td>EFI_ABORTED</td>
<td>Current operation is aborted by user.</td>
</tr>
<tr>
<td>EFI_NETWORK_UNREACHABLE</td>
<td>An ICMP network unreachable error packet was received.</td>
</tr>
<tr>
<td>EFI_PROTOCOL_UNREACHABLE</td>
<td>An ICMP host unreachable error packet was received.</td>
</tr>
<tr>
<td>EFI_NETWORK_UNREACHABLE</td>
<td>An ICMP protocol unreachable error packet was received.</td>
</tr>
<tr>
<td>EFI_NETWORK_UNREACHABLE</td>
<td>An ICMP port unreachable error packet was received.</td>
</tr>
<tr>
<td>EFI_ICMP_ERROR</td>
<td>Some other ICMP ERROR packet was received.</td>
</tr>
<tr>
<td>EFI_TIMEOUT</td>
<td>No responses were received from the MTFTPv4 server.</td>
</tr>
<tr>
<td>EFI_TFTP_ERROR</td>
<td>An MTFTPv4 ERROR packet was received.</td>
</tr>
<tr>
<td>EFI_DEVICE_ERROR</td>
<td>An unexpected network error or system error occurred.</td>
</tr>
<tr>
<td>EFI_NO_MEDIA</td>
<td>There was a media error.</td>
</tr>
</tbody>
</table>

EFI_MTFTP4_CHECK_PACKET is a callback function that is provided by the caller to intercept the EFI_MTFTP_OPCODE_DATA or EFI_MTFTP_OPCODE_DATA8 packets processed in the EFI_MTFTP_PROTOCOL.ReadFile() function, and alternatively to intercept EFI_MTFTP_OPCODE_OACK or EFI_MTFTP_OPCODE_ERROR packets during a call to EFI_MTFTP_PROTOCOL.ReadFile(), WriteFile() or ReadDirectory(). Whenever an MTFTPv4 packet with the type described above is received from a server, the EFI MTFTPv4 Protocol driver will call
**EFI_MTFTP4_CHECK_PACKET** function to let the caller have an opportunity to process this packet. Any status code other than **EFI_SUCCESS** that is returned from this function will abort the transfer process.

```c
EFI_MTFTP4_TIMEOUT_CALLBACK
```

This Pointer to the **EFI_MTFTP4_PROTOCOL** instance.

Token The token that is provided in the

**EFI_MTFTP4_PROTOCOL.ReadFile()** or **EFI_MTFTP4_PROTOCOL.WriteFile()** or **EFI_MTFTP4_PROTOCOL.ReadDirectory()** functions by the caller. Type **EFI_MTFTP4_TOKEN** is defined in **EFI_MTFTP4_PROTOCOL.ReadFile()**.

**EFI_MTFTP4_TIMEOUT_CALLBACK** is a callback function that the caller provides to capture the timeout event in the **EFI_MTFTP4_PROTOCOL.ReadFile()**, **EFI_MTFTP4_PROTOCOL.WriteFile()** or **EFI_MTFTP4_PROTOCOL.ReadDirectory()** functions. Whenever a timeout occurs, the EFI MTFTPv4 Protocol driver will call the **EFI_MTFTP4_TIMEOUT_CALLBACK** function to notify the caller of the timeout event. Any status code other than **EFI_SUCCESS** that is returned from this function will abort the current download process.

```c
EFI_MTFTP4_PACKET_NEEDED
```

This Pointer to the **EFI_MTFTP4_PROTOCOL** instance.

Token The token provided in the

**EFI_MTFTP4_PROTOCOL.WriteFile()** by the caller.

Length Indicates the length of the raw data wanted on input, and the length the data available on output.

Buffer Pointer to the buffer where the data is stored.
EFI_MTFTP4_PACKET_NEEDED is a callback function that the caller provides to feed data to the EFI_MTFTP4_PROTOCOL.WriteFile() function. EFI_MTFTP4_PACKET_NEEDED provides another mechanism for the caller to provide data to upload other than a static buffer. The EFI MTFTP4 Protocol driver always calls EFI_MTFTP4_PACKET_NEEDED to get packet data from the caller if no static buffer was given in the initial call to EFI_MTFTP4_PROTOCOL.WriteFile() function. Setting *Length to zero signals the end of the session. Returning a status code other than EFI_SUCCESS aborts the session.

Status Codes Returned

<table>
<thead>
<tr>
<th>Status Code</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>EFI_SUCCESS</td>
<td>The data file is being downloaded.</td>
</tr>
<tr>
<td>EFI_INVALID_PARAMETER</td>
<td>One or more of the parameters is not valid.</td>
</tr>
<tr>
<td></td>
<td>• This is NULL.</td>
</tr>
<tr>
<td></td>
<td>• Token is NULL.</td>
</tr>
<tr>
<td></td>
<td>• Token.Filename is NULL.</td>
</tr>
<tr>
<td></td>
<td>• Token.OptionCount is not zero and Token.OptionList is NULL.</td>
</tr>
<tr>
<td></td>
<td>• One or more options in Token.OptionList have wrong format.</td>
</tr>
<tr>
<td></td>
<td>• Token.Buffer and Token.CheckPacket are both NULL.</td>
</tr>
<tr>
<td></td>
<td>• One or more IPv4 addresses in Token.OverrideData are not valid unicast IPv4 addresses if Token.OverrideData is not NULL and the addresses are not set to all zero.</td>
</tr>
<tr>
<td>EFI_UNSUPPORTED</td>
<td>One or more options in the Token.OptionList are in the unsupported list of structure EFI_MTFTP4_MODE_DATA</td>
</tr>
<tr>
<td>EFI_NOT_STARTED</td>
<td>The EFI MTFTP4 Protocol driver has not been started.</td>
</tr>
<tr>
<td>EFI_NO_MAPPING</td>
<td>When using a default address, configuration (DHCP, BOOTP, RARP, etc.) is not finished yet.</td>
</tr>
<tr>
<td>EFI_ALREADY_STARTED</td>
<td>This Token is being used in another MTFTPv4 session.</td>
</tr>
<tr>
<td>EFI_ACCESS_DENIED</td>
<td>The previous operation has not completed yet.</td>
</tr>
<tr>
<td>EFI_OUT_OF_RESOURCES</td>
<td>Required system resources could not be allocated.</td>
</tr>
<tr>
<td>EFIDEVICE_ERROR</td>
<td>An unexpected network error or system error occurred.</td>
</tr>
<tr>
<td>EFI_NO_MEDIA</td>
<td>There was a media error.</td>
</tr>
</tbody>
</table>

EFI_MTFTP4_PROTOCOL.WriteFile()

Summary

Sends a data file to an MTFTPv4 server. May be unsupported in some EFI implementations.
Prototype

```c
typedef EFI_STATUS
(EFI_API *EFI_MTFTP4_WRITE_FILE)(
    IN EFI_MTFTP4_PROTOCOL *This,
    IN EFI_MTFTP4_TOKEN *Token
);
```

Parameters

- **This**: Pointer to the `EFI_MTFTP4_PROTOCOL` instance.
- **Token**: Pointer to the token structure to provide the parameters that are used in this function. Type `EFI_MTFTP4_TOKEN` is defined in `EFI_MTFTP4_PROTOCOL.ReadFile()`.

Description

The `WriteFile()` function is used to initialize an uploading operation with the given option list and optionally wait for completion. If one or more of the options is not supported by the server, the unsupported options are ignored and a standard TFTP process starts instead. When the upload process completes, whether successfully or not, `Token.Event` is signaled, and the EFI MTFTPv4 Protocol driver updates `Token.Status`.

The caller can supply the data to be uploaded in the following two modes:

- Through the user-provided buffer
- Through a callback function

With the user-provided buffer, the `Token.BufferSize` field indicates the length of the buffer, and the driver will upload the data in the buffer. With an `EFI_MTFTP4_PACKET_NEEDED` callback function, the driver will call this callback function to get more data from the user to upload. See the definition of `EFI_MTFTP4_PACKET_NEEDED` for more information. These two modes cannot be used at the same time. The callback function will be ignored if the user provides the buffer.

Status Codes Returned

<table>
<thead>
<tr>
<th>Status Code</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>EFI_SUCCESS</td>
<td>The upload session has started.</td>
</tr>
<tr>
<td>EFI_UNSUPPORTED</td>
<td>The operation is not supported by this implementation.</td>
</tr>
<tr>
<td>EFI_INVALID_PARAMETER</td>
<td>One or more of the following conditions is <strong>TRUE</strong>.</td>
</tr>
<tr>
<td></td>
<td>- <strong>This</strong> is <strong>NULL</strong>.</td>
</tr>
<tr>
<td></td>
<td>- <strong>Token</strong> is <strong>NULL</strong>.</td>
</tr>
<tr>
<td></td>
<td>- <strong>Token.Filename</strong> is <strong>NULL</strong>.</td>
</tr>
<tr>
<td></td>
<td>- <strong>Token.OptionCount</strong> is not zero and <strong>Token.OptionList</strong> is <strong>NULL</strong>.</td>
</tr>
<tr>
<td></td>
<td>- One or more options in <strong>Token.OptionList</strong> have wrong format.</td>
</tr>
<tr>
<td></td>
<td>- <strong>Token.Buffer</strong> and <strong>Token.PacketNeeded</strong> are both <strong>NULL</strong>.</td>
</tr>
<tr>
<td></td>
<td>- One or more IPv4 addresses in <strong>Token.OverrideData</strong> are not valid unicast IPv4 addresses if <strong>Token.OverrideData</strong> is not <strong>NULL</strong> and the addresses are not set to all zero.</td>
</tr>
</tbody>
</table>
**EFI_MTFTP4_PROTOCOL.ReadDirectory()**

**Summary**
Downloads a data file “directory” from an MTFTPv4 server. May be unsupported in some EFI implementations.

**Prototype**
```c
typedef EFI_STATUS
(EIFIAPI *EFI_MTFTP4_READ_DIRECTORY)(
  IN EFI_MTFTP4_PROTOCOL *This,
  IN EFI_MTFTP4_TOKEN *Token);
```

**Parameters**
- **This** Pointer to the EFI_MTFTP4_PROTOCOL instance.
- **Token** Pointer to the token structure to provide the parameters that are used in this function. Type EFI_MTFTP4_TOKEN is defined in EFI_MTFTP4_PROTOCOL.ReadFile().

**Description**
The **ReadDirectory()** function is used to return a list of files on the MTFTPv4 server that are logically (or operationally) related to **Token.Filename**. The directory request packet that is sent to the server is built with the option list that was provided by caller, if present.

The file information that the server returns is put into either of the following locations:
- A fixed buffer that is pointed to by **Token.Buffer**
- A download service function that is pointed to by **Token.CheckPacket**

If both **Token.Buffer** and **Token.CheckPacket** are used, then **Token.CheckPacket** will be called first. If the call is successful, the packet will be stored in **Token.Buffer**.

The returned directory listing in the **Token.Buffer** or **EFI_MTFTP4_PACKET** consists of a list of two or three variable-length ASCII strings, each terminated by a null character, for each file in the directory. If the multicast option is involved, the first field of each directory entry is the static multicast IP address and
UDP port number that is associated with the file name. The format of the field is ip:ip:ip:port. If the multicast option is not involved, this field and its terminating null character are not present.

The next field of each directory entry is the file name and the last field is the file information string. The information string contains the file size and the create/modify timestamp. The format of the information string is filesize yyyy-mm-dd hh:mm:ss:ffff. The timestamp is Coordinated Universal Time (UTC; also known as Greenwich Mean Time [GMT]).

**Status Codes Returned**

<table>
<thead>
<tr>
<th>Code</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>EFI_SUCCESS</td>
<td>The MTFTPv4 related file &quot;directory&quot; has been downloaded.</td>
</tr>
<tr>
<td>EFI_UNSUPPORTED</td>
<td>The EFI MTFTPv4 Protocol driver does not support this function.</td>
</tr>
<tr>
<td>EFI_INVALID_PARAMETER</td>
<td>One or more of these conditions is TRUE:</td>
</tr>
<tr>
<td></td>
<td>• This is NULL.</td>
</tr>
<tr>
<td></td>
<td>• Token is NULL.</td>
</tr>
<tr>
<td></td>
<td>• Token.filename is NULL.</td>
</tr>
<tr>
<td></td>
<td>• Token.OptionCount is not zero and Token.OptionList is NULL.</td>
</tr>
<tr>
<td></td>
<td>• One or more options in Token.OptionList have wrong format.</td>
</tr>
<tr>
<td></td>
<td>• One or more IPv4 addresses in Token.OverrideData are not valid unicast IPv4 addresses if Token.OverrideData is not NULL and the addresses are not set to all zero.</td>
</tr>
<tr>
<td>EFI_UNSUPPORTED</td>
<td>One or more options in the Token.OptionList are in the unsupported list of structure EFI_MTFTP4_MODE_DATA.</td>
</tr>
<tr>
<td>EFI_NOT_STARTED</td>
<td>The EFI MTFTPv4 Protocol driver has not been started.</td>
</tr>
<tr>
<td>EFI_NO_MAPPING</td>
<td>When using a default address, configuration (DHCP, BOOTP, RARP, etc.) is not finished yet.</td>
</tr>
<tr>
<td>EFI_ALREADY_STARTED</td>
<td>This Token is already being used in another MTFTPv4 session.</td>
</tr>
<tr>
<td>EFI_OUT_OF_RESOURCES</td>
<td>Required system resources could not be allocated.</td>
</tr>
<tr>
<td>EFI_ACCESS_DENIED</td>
<td>The previous operation has not completed yet.</td>
</tr>
<tr>
<td>EFI_DEVICE_ERROR</td>
<td>An unexpected network error or system error occurred.</td>
</tr>
<tr>
<td>EFI_NO_MEDIA</td>
<td>There was a media error.</td>
</tr>
</tbody>
</table>

**EFI_MTFTP4_PROTOCOL.POLL()**

**Summary**

Polls for incoming data packets and processes outgoing data packets.
Prototype

```c
typedef
EFI_STATUS
(EFIAPI *EFI_MTFTP4_POLL) (
    IN EFI_MTFTP4_PROTOCOL *This
);
```

Parameters

- **This**: Pointer to the `EFI_MTFTP4_PROTOCOL` instance.

Description

The `Poll()` function can be used by network drivers and applications to increase the rate that data packets are moved between the communications device and the transmit and receive queues.

In some systems, the periodic timer event in the managed network driver may not poll the underlying communications device fast enough to transmit and/or receive all data packets without missing incoming packets or dropping outgoing packets. Drivers and applications that are experiencing packet loss should try calling the `Poll()` function more often.

Status Codes Returned

<table>
<thead>
<tr>
<th>Status Code</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>EFI_SUCCESS</td>
<td>Incoming or outgoing data was processed.</td>
</tr>
<tr>
<td>EFI_NOT_STARTED</td>
<td>This EFI MTFTPv4 Protocol instance has not been started.</td>
</tr>
<tr>
<td>EFI_NO_MAPPING</td>
<td>When using a default address, configuration (DHCP, BOOTP, RARP, etc.) is</td>
</tr>
<tr>
<td></td>
<td>not finished yet.</td>
</tr>
<tr>
<td>EFI_INVALID_PARAMETER</td>
<td><strong>This</strong> is <strong>NULL</strong>.</td>
</tr>
<tr>
<td>EFI_DEVICE_ERROR</td>
<td>An unexpected system or network error occurred.</td>
</tr>
<tr>
<td>EFI_TIMEOUT</td>
<td>Data was dropped out of the transmit and/or receive queue. Consider</td>
</tr>
<tr>
<td></td>
<td>increasing the polling rate.</td>
</tr>
</tbody>
</table>

30.4 EFI MTFTPv6 Protocol

This section defines the EFI MTFTPv6 Protocol interface that is built upon the EFI UDPv6 Protocol.

30.4.1 MTFTP6 Service Binding Protocol

**EFI_MTFTP6_SERVICE_BINDING_PROTOCOL**

Summary

The EFI MTFTPv6 Service Binding Protocol is used to locate communication devices that are supported by an EFI MTFTPv6 Protocol driver and to create and destroy instances of the EFI MTFTPv6 Protocol child instance that can use the underlying communications device.
GUID

```c
#define EFI_MTFTP6_SERVICE_BINDING_PROTOCOL_GUID \
{0xd9760ff3,0x3cca,0x4267,\ 
{0x80,0xf9,0x75,0x27,0xfa,0xfa,0x42,0x23}}
```

Description

A network application or driver that requires MTFTPv6 I/O services can use one of the protocol handler services, such as `BS->LocateHandleBuffer()`, to search for devices that publish an EFI MTFTPv6 Service Binding Protocol GUID. Each device with a published EFI MTFTPv6 Service Binding Protocol GUID supports the EFI MTFTPv6 Protocol service and may be available for use.

After a successful call to the `EFI_MTFTP6_SERVICE_BINDING_PROTOCOL.CreateChild()` function, the newly created child EFI MTFTPv6 Protocol driver instance is in the un-configured state; it is not ready to transfer data.

Before a network application terminates execution, every successful call to the `EFI_MTFTP6_SERVICE_BINDING_PROTOCOL.CreateChild()` function must be matched with a call to the `EFI_MTFTP6_SERVICE_BINDING_PROTOCOL.DestroyChild()` function.

Each instance of the EFI MTFTPv6 Protocol driver can support one file transfer operation at a time. To download two files at the same time, two instances of the EFI MTFTPv6 Protocol driver need to be created.

30.4.2 MTFTP6 Protocol

**EFI_MTFTP6_PROTOCOL**

Summary

The EFI MTFTPv6 Protocol provides basic services for client-side unicast and/or multicast TFTP operations.

GUID

```c
#define EFI_MTFTP6_PROTOCOL_GUID \
{0xbf0a78ba,0xec29,0x49cf,\ 
{0xa1,0xc9,0x7a,0xe5,0x4e,0xab,0x6a,0x51}}
```

Protocol Interface Structure

```c
typedef struct _EFI_MTFTP6_PROTOCOL {
  EFI_MTFTP6_GET_MODE_DATA     GetModeData;
  EFI_MTFTP6_CONFIGURE         Configure;
  EFI_MTFTP6_GET_INFO          GetInfo;
  EFI_MTFTP6_PARSE_OPTIONS     ;
  EFI_MTFTP6_READ_FILE         ReadFile;
  EFI_MTFTP6_WRITE_FILE        WriteFile;
  EFI_MTFTP6_READ_DIRECTORY    ReadDirectory;
  EFI_MTFTP6_POLL              Poll;
} EFI_MTFTP6_PROTOCOL;
```
Parameters

**GetModeData**
Reads the current operational settings. See the `GetModeData()` function description.

**Configure**
Initializes, changes, or resets the operational settings for this instance of the EFI MTFTPv6 Protocol driver. See the `Configure()` function description.

**GetInfo**
Retrieves information about a file from an MTFTPv6 server. See the `GetInfo()` function description.

Parses the options in an MTFTPv6 OACK (options acknowledgement) packet. See the () function description.

**ReadFile**
Downloads a file from an MTFTPv6 server. See the `ReadFile()` function description.

**WriteFile**
Uploads a file to an MTFTPv6 server. This function may be unsupported in some EFI implementations. See the `WriteFile()` function description.

**ReadDirectory**
Downloads a related file directory from an MTFTPv6 server. This function may be unsupported in some EFI implementations. See the `ReadDirectory()` function description.

**Poll**
Polls for incoming data packets and processes outgoing data packets. See the `Poll()` function description.

Description

The `EFI_MTFTP6_PROTOCOL` is designed to be used by UEFI drivers and applications to transmit and receive data files. The EFI MTFTPv6 Protocol driver uses the underlying EFI UDPv6 Protocol driver and EFI IPv6 Protocol driver.

**EFI_MTFTP6_PROTOCOL.GetModeData()**

**Summary**
Read the current operational settings.

**Prototype**

```c
typedef
EFI_STATUS
(EIFIAPI *EFI_MTFTP6_GET_MODE_DATA)(
   IN EFI_MTFTP6_PROTOCOL   *This,
   OUT EFI_MTFTP6_MODE_DATA   *ModeData
);
```

**Parameters**

*This*

Pointer to the `EFI_MTFTP6_PROTOCOL` instance.
**ModeData**

The buffer in which the EFI MTFTPv6 Protocol driver mode data is returned. Type `EFI_MTFTP6_MODE_DATA` is defined in "Related Definitions" below.

**Description**

The `GetModeData()` function reads the current operational settings of this EFI MTFTPv6 Protocol driver instance.

**Related Definitions**

```c
//******************************************************************************
// EFI_MTFTP6_MODE_DATA
//******************************************************************************
typedef struct {
  EFI_MTFTP6_CONFIG_DATA ConfigData;
  UINT8 SupportedOptionCount;
  UINT8 **SupportedOptions;
} EFI_MTFTP6_MODE_DATA;
```

`ConfigData` The configuration data of this instance. Type `EFI_MTFTP6_CONFIG_DATA` is defined below.

`SupportedOptionCount` The number of option strings in the following `SupportedOptions` array.

`SupportedOptions` An array of null-terminated ASCII option strings that are recognized and supported by this EFI MTFTPv6 Protocol driver implementation. The buffer is read only to the caller and the caller should NOT free the buffer.

The `EFI_MTFTP6_MODE_DATA` structure describes the operational state of this instance.

```c
//******************************************************************************
// EFI_MTFTP6_CONFIG_DATA
//******************************************************************************
typedef struct {
  EFI_IPv6_ADDRESS StationIp;
  UINT16 LocalPort;
  EFI_IPv6_ADDRESS ServerIp;
  UINT16 InitialServerPort;
  UINT16 TryCount;
  UINT16 TimeoutValue;
} EFI_MTFTP6_CONFIG_DATA;
```

`StationIp` The local IP address to use. Set to zero to let the underlying IPv6 driver choose a source address. If not zero it must be one of the configured IP addresses in the underlying IPv6 driver.
LocalPort
Local port number. Set to zero to use the automatically assigned port number.

ServerIp
The IP address of the MTFTPv6 server.

InitialServerPort
The initial MTFTPv6 server port number. Request packets are sent to this port. This number is almost always 69 and using zero defaults to 69.

TryCount
The number of times to transmit MTFTPv6 request packets and wait for a response.

TimeoutValue
The number of seconds to wait for a response after sending the MTFTPv6 request packet.

The **EFI_MTFTP6_CONFIG_DATA** structure is used to retrieve and change MTFTPv6 session parameters.

### Status Codes Returned

<table>
<thead>
<tr>
<th>Code</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>EFI_SUCCESS</td>
<td>The configuration data was successfully returned.</td>
</tr>
<tr>
<td>EFI_OUT_OF_RESOURCES</td>
<td>The required mode data could not be allocated.</td>
</tr>
<tr>
<td>EFI_INVALID_PARAMETER</td>
<td>This is <strong>NULL</strong> or <strong>ModeData</strong> is <strong>NULL</strong>.</td>
</tr>
</tbody>
</table>

**EFI_MTFTP6_PROTOCOL.Configure()**

**Summary**

Initializes, changes, or resets the default operational setting for this EFI MTFTPv6 Protocol driver instance.

**Prototype**

```c
typedef
EFI_STATUS
(EFIAPI *EFI_MTFTP6_CONFIGURE)(
    IN EFI_MTFTP6_PROTOCOL *This,
    IN EFI_MTFTP6_CONFIG_DATA *MtftpConfigData OPTIONAL
);
```

**Parameters**

- **This**
  Pointer to the **EFI_MTFTP6_PROTOCOL** instance.
- **MtftpConfigData**
  Pointer to the configuration data structure. Type **EFI_MTFTP6_CONFIG_DATA** is defined in **EFI_MTFTP6_PROTOCOL.GetModeData()**.

**Description**

The **Configure()** function is used to set and change the configuration data for this EFI MTFTPv6 Protocol driver instance. The configuration data can be reset to startup defaults by calling **Configure()** with **MtftpConfigData** set to **NULL**. Whenever the instance is reset, any pending operation is aborted. By changing the EFI MTFTPv6 Protocol driver instance configuration data, the client can connect to different MTFTPv6 servers. The configuration parameters in **MtftpConfigData** are used as the default parameters in later MTFTPv6 operations and can be overridden in later operations.
### Status Codes Returned

<table>
<thead>
<tr>
<th>Code</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>EFI_SUCCESS</td>
<td>The EFI MTFTPv6 Protocol instance was configured successfully.</td>
</tr>
<tr>
<td>EFI_INVALID_PARAMETER</td>
<td>One or more following conditions are TRUE:</td>
</tr>
<tr>
<td></td>
<td>• This is NULL</td>
</tr>
<tr>
<td></td>
<td>• MtftpConfigData.StationIp is neither zero nor one of the configured IP addresses in the underlying IPv6 driver.</td>
</tr>
<tr>
<td></td>
<td>• MtftpConfigData.ServerIp is not a valid IPv6 unicast address.</td>
</tr>
<tr>
<td>EFI_ACCESS_DENIED</td>
<td>• The configuration could not be changed at this time because there is some MTFTP background operation in progress.</td>
</tr>
<tr>
<td></td>
<td>• MtftpConfigData.LocalPort is already in use.</td>
</tr>
<tr>
<td>EFI_NO_MAPPING</td>
<td>The underlying IPv6 driver was responsible for choosing a source address for this instance, but no source address was available for use.</td>
</tr>
<tr>
<td>EFI_OUT_OF_RESOURCES</td>
<td>The EFI MTFTPv6 Protocol driver instance data could not be allocated.</td>
</tr>
<tr>
<td>EFI_DEVICE_ERROR</td>
<td>An unexpected system or network error occurred. The EFI MTFTPv6 Protocol driver instance is not configured.</td>
</tr>
</tbody>
</table>

### EFI_MTFTP6_PROTOCOL.GetInfo()

**Summary**

Get information about a file from an MTFTPv6 server.

**Prototype**

```c
typedef
EFI_STATUS
(EFIAPI *EFI_MTFTP6_GET_INFO)(
  IN EFI_MTFTP6_PROTOCOL    *This,
  IN EFI_MTFTP6_OVERRIDE_DATA  *OverrideData  OPTIONAL,
  IN UINT8           *Filename,
  IN UINT8           *ModeStr     OPTIONAL,
  IN UINT8          OptionCount,
  IN EFI_MTFTP6_OPTION      *OptionList OPTIONAL,
  OUT UINT32           *PacketLength,
  OUT EFI_MTFTP6_PACKET     **Packet OPTIONAL
);
```

**Parameters**

- **This**
  - Pointer to the **EFI_MTFTP6_PROTOCOL** instance.

- **OverrideData**
  - Data that is used to override the existing parameters. If **NULL**, the default parameters that were set in the **EFI_MTFTP6_PROTOCOL.Configure()** function are used. Type **EFI_MTFTP6_OVERRIDE_DATA** is defined in "Related Definitions" below.

- **Filename**
  - Pointer to an null-terminated ASCII file name string.
ModeStr  
Pointer to an null-terminated ASCII mode string. If NULL, octet will be used.

OptionCount  
Number of option/value string pairs in OptionList.

OptionList  
Pointer to array of option/value string pairs. Ignored if OptionCount is zero. Type EFI_MTFTP6_OPTION is defined in "Related Definitions" below.

PacketLength  
The number of bytes in the returned packet.

Packet  
The pointer to the received packet. This buffer must be freed by the caller. Type EFI_MTFTP6_PACKET is defined in "Related Definitions" below.

Description
The GetInfo() function assembles an MTFTPv6 request packet with options, sends it to the MTFTPv6 server, and may return an MTFTPv6 OACK, MTFTPv6 ERROR, or ICMP ERROR packet. Retries occur only if no response packets are received from the MTFTPv6 server before the timeout expires.

Related Definitions

```c
typedef struct {
    EFI_IPv6_ADDRESS ServerIp;
    UINT16 ServerPort;
    UINT16 TryCount;
    UINT16 TimeoutValue;
} EFI_MTFTP6_OVERRIDE_DATA;
```

ServerIp  
IP address of the MTFTPv6 server. If set to all zero, the value that was set by the EFI_MTFTP6_PROTOCOL.Configure() function will be used.

ServerPort  
MTFTPv6 server port number. If set to zero, it will use the value that was set by the EFI_MTFTP6_PROTOCOL.Configure() function.

TryCount  
Number of times to transmit MTFTPv6 request packets and wait for a response. If set to zero, the value that was set by the EFI_MTFTP6_PROTOCOL.Configure() function will be used.

TimeoutValue  
Number of seconds to wait for a response after sending the MTFTPv6 request packet. If set to zero, the value that was set by the EFI_MTFTP6_PROTOCOL.Configure() function will be used.

The EFI_MTFTP6_OVERRIDE_DATA structure is used to override the existing parameters that were set by the EFI_MTFTP6_PROTOCOL.Configure() function.
typedef struct {
    UINT8 *OptionStr;
    UINT8 *ValueStr;
} EFI_MTFTP6_OPTION;

OptionStr
Pointer to the null-terminated ASCII MTFTPv6 option string.

ValueStr
Pointer to the null-terminated ASCII MTFTPv6 value string.

#pragma pack(1)

typedef union {
    UINT16 OpCode;
    EFI_MTFTP6_REQ_HEADER Rrq;
    EFI_MTFTP6_REQ_HEADER Wrq;
    EFI_MTFTP6_OACK_HEADER Oack;
    EFI_MTFTP6_DATA_HEADER Data;
    EFI_MTFTP6_ACK_HEADER Ack;
    // This field should be ignored and treated as reserved.
    EFI_MTFTP6_DATA8_HEADER Data8;
    // This field should be ignored and treated as reserved.
    EFI_MTFTP6_ACK8_HEADER Ack8;
    EFI_MTFTP6_ERROR_HEADER Error;
} EFI_MTFTP6_PACKET;

typedef struct {
    UINT16 OpCode;
    UINT8 Filename[1];
} EFI_MTFTP6_REQ_HEADER;
typedef struct {
    UINT16 OpCode;
    UINT8 Data[1];
} EFI_MTFTP6_OACK_HEADER;

typedef struct {
    UINT16 OpCode;
    UINT16 Block;
    UINT8 Data[1];
} EFI_MTFTP6_DATA_HEADER;

typedef struct {
    UINT16 OpCode;
    UINT16 Block[1];
} EFI_MTFTP6_ACK_HEADER;

typedef struct {
    UINT16 OpCode;
    UINT64 Block;
    UINT8 Data[1];
} EFI_MTFTP6_DATA8_HEADER;

typedef struct {
    UINT16 OpCode;
    UINT64 Block[1];
} EFI_MTFTP6_ACK8_HEADER;

// EFI_MTFTP6_ERROR_HEADER

// This field should be ignored and treated as reserved.
typedef struct {
    UINT16    OpCode;
    UINT16    ErrorCode;
    UINT8     ErrorMessage[1];
} EFI_MTFTP6_ERROR_HEADER;

#pragma pack()
<table>
<thead>
<tr>
<th>EFI_MTFTP6_PACKET</th>
<th>OpCode</th>
<th>Type of packets as defined by the MTFTPv6 packet opcodes. Opcode values are defined below.</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Rrq, Wrq</td>
<td>Read request or write request packet header. See the description for EFI_MTFTP6_REQ_HEADER below in this table.</td>
</tr>
<tr>
<td></td>
<td>Oack</td>
<td>Option acknowledge packet header. See the description for EFI_MTFTP6_OACK_HEADER below in this table.</td>
</tr>
<tr>
<td></td>
<td>Data</td>
<td>Data packet header. See the description for EFI_MTFTP6_DATA_HEADER below in this table.</td>
</tr>
<tr>
<td></td>
<td>Ack</td>
<td>Acknowledgement packet header. See the description for EFI_MTFTP6_ACK_HEADER below in this table.</td>
</tr>
<tr>
<td></td>
<td>Data8</td>
<td>This field should be ignored and treated as reserved. Data packet header with big block number. See the description for EFI_MTFTP6_DATA8_HEADER below in this table.</td>
</tr>
<tr>
<td></td>
<td>Ack8</td>
<td>This field should be ignored and treated as reserved. Acknowledgement header with big block number. See the description for EFI_MTFTP6_ACK8_HEADER below in this table.</td>
</tr>
<tr>
<td></td>
<td>Error</td>
<td>Error packet header. See the description for EFI_MTFTP6_ERROR_HEADER below in this table.</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>EFI_MTFTP6_REQ_HEADER</th>
<th>OpCode</th>
<th>For this packet type, OpCode = EFI_MTFTP6_OPCODE_RRQ for a read request or OpCode = EFI_MTFTP6_OPCODE_WRQ for a write request.</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Filename</td>
<td>The file name to be downloaded or uploaded.</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>EFI_MTFTP6_OACK_HEADER</th>
<th>OpCode</th>
<th>For this packet type, OpCode = EFI_MTFTP6_OPCODE_OACK</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Data</td>
<td>The option strings in the option acknowledgement packet.</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>EFI_MTFTP6_DATA_HEADER</th>
<th>OpCode</th>
<th>For this packet type, OpCode = EFI_MTFTP6_OPCODE_DATA</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Block</td>
<td>Block number of this data packet.</td>
</tr>
<tr>
<td></td>
<td>Data</td>
<td>The content of this data packet.</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>EFI_MTFTP6_ACK_HEADER</th>
<th>OpCode</th>
<th>For this packet type, OpCode = EFI_MTFTP6_OPCODE_ACK</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Block</td>
<td>The block number of the data packet that is being acknowledged.</td>
</tr>
</tbody>
</table>
MTFTP Packet OpCode Descriptions

<table>
<thead>
<tr>
<th>MTFTP Packet OpCode</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>EFI_MTFTP6_OPCODE_RRQ</td>
<td>The MTFTPv6 packet is a read request.</td>
</tr>
<tr>
<td>EFI_MTFTP6_OPCODE_WRQ</td>
<td>The MTFTPv6 packet is a write request.</td>
</tr>
<tr>
<td>EFI_MTFTP6_OPCODE_DATA</td>
<td>The MTFTPv6 packet is a data packet.</td>
</tr>
</tbody>
</table>

Following is a description of the fields in the above definition.
Table 30-4 MTFTP ERROR Packet ErrorCode Descriptions

<table>
<thead>
<tr>
<th>MTFTP ERROR Packet ErrorCodes</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>EFI_MTFTP6_ERRORCODE_NOT_DEFINED</td>
<td>The error code is not defined. See the error message in the packet (if any) for details.</td>
</tr>
<tr>
<td>EFI_MTFTP6_ERRORCODE_FILE_NOT_FOUND</td>
<td>The file was not found.</td>
</tr>
<tr>
<td>EFI_MTFTP6_ERRORCODE_ACCESS_VIOLATION</td>
<td>There was an access violation.</td>
</tr>
<tr>
<td>EFI_MTFTP6_ERRORCODE_DISK_FULL</td>
<td>The disk was full or its allocation was exceeded.</td>
</tr>
<tr>
<td>EFI_MTFTP6_ERRORCODE_ILLEGAL_OPERATION</td>
<td>The MTFTPv6 operation was illegal.</td>
</tr>
<tr>
<td>EFI_MTFTP6_ERRORCODE_UNKNOWN_TRANSFER_ID</td>
<td>The transfer ID is unknown.</td>
</tr>
<tr>
<td>EFI_MTFTP6_ERRORCODE_FILE_ALREADY_EXISTS</td>
<td>The file already exists.</td>
</tr>
<tr>
<td>EFI_MTFTP6_ERRORCODE_NO_SUCH_USER</td>
<td>There is no such user.</td>
</tr>
<tr>
<td>EFI_MTFTP6_ERRORCODE_REQUEST_DENIED</td>
<td>The request has been denied due to option negotiation.</td>
</tr>
</tbody>
</table>
Status Codes Returned

<table>
<thead>
<tr>
<th>Status Code</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>EFI_SUCCESS</td>
<td>An MTFTPv6 OACK packet was received and is in the Packet.</td>
</tr>
<tr>
<td>EFI_INVALID_PARAMETER</td>
<td>One or more of the following conditions is TRUE:</td>
</tr>
<tr>
<td></td>
<td>• This is NULL.</td>
</tr>
<tr>
<td></td>
<td>• Filename is NULL.</td>
</tr>
<tr>
<td></td>
<td>• OptionCount is not zero and OptionList is NULL.</td>
</tr>
<tr>
<td></td>
<td>• One or more options in OptionList have wrong format.</td>
</tr>
<tr>
<td></td>
<td>• PacketLength is NULL.</td>
</tr>
<tr>
<td></td>
<td>• OverrideData.ServerIp is not a valid unicast IPv6 address and not set to all zero.</td>
</tr>
<tr>
<td>EFI_UNSUPPORTED</td>
<td>One or more options in the OptionList are unsupported by this implementation.</td>
</tr>
<tr>
<td>EFI_NOT_STARTED</td>
<td>The EFI MTFTPv6 Protocol driver has not been started.</td>
</tr>
<tr>
<td>EFI_NO_MAPPING</td>
<td>The underlying IPv6 driver was responsible for choosing a source address for this instance, but no source address was available for use.</td>
</tr>
<tr>
<td>EFI_ACCESS_DENIED</td>
<td>The previous operation has not completed yet.</td>
</tr>
<tr>
<td>EFI_OUT_OF_RESOURCES</td>
<td>Required system resources could not be allocated.</td>
</tr>
<tr>
<td>EFI_TFTP_ERROR</td>
<td>An MTFTPv6 ERROR packet was received and is in the Packet.</td>
</tr>
<tr>
<td>EFI_NETWORK_UNREACHABLE</td>
<td>An ICMP network unreachable error packet was received and the Packet is set to NULL.</td>
</tr>
<tr>
<td>EFI_NETWORK_UNREACHABLE</td>
<td>An ICMP host unreachable error packet was received and the Packet is set to NULL.</td>
</tr>
<tr>
<td>EFI_NETWORK_UNREACHABLE</td>
<td>An ICMP protocol unreachable error packet was received and the Packet is set to NULL.</td>
</tr>
<tr>
<td>EFI_PROTOCOL_ERROR</td>
<td>Some other ICMP ERROR packet was received and the Packet is set to NULL.</td>
</tr>
<tr>
<td>EFI_TIMEOUT</td>
<td>No responses were received from the MTFTPv6 server.</td>
</tr>
<tr>
<td>EFI_DEVICE_ERROR</td>
<td>An unexpected network error or system error occurred.</td>
</tr>
</tbody>
</table>

EFI_MTFTP6_PROTOCOL.ParseOptions()

Summary

Parse the options in an MTFTPv6 OACK packet.
Prototype

```c
typedef EFI_STATUS
(EIFI_API *EFI_MTFTP6_PARSE_OPTIONS)(
    IN EFI_MTFTP6_PROTOCOL *This,
    IN UINT32 PacketLen,
    IN EFI_MTFTP6_PACKET  *Packet,
    OUT UINT32       *OptionCount,
    OUT EFI_MTFTP6_OPTION  **OptionList OPTIONAL
);
```

Parameters

- **This**
  Pointer to the EFI_MTFTP6_PROTOCOL instance.
- **PacketLen**
  Length of the OACK packet to be parsed.
- **Packet**
  Pointer to the OACK packet to be parsed. Type EFI_MTFTP6_PACKET is defined in EFI_MTFTP6_PROTOCOL.GetInfo().
- **OptionCount**
  Pointer to the number of options in the following OptionList.
- **OptionList**
  Pointer to EFI_MTFTP6_OPTION storage. Each pointer in the OptionList points to the corresponding MTFTP option buffer in the Packet. Call the EFI Boot Service FreePool() to release the OptionList if the options in this OptionList are not needed any more. Type EFI_MTFTP6_OPTION is defined in EFI_MTFTP6_PROTOCOL.GetInfo().

Description

The ParseOptions() function parses the option fields in an MTFTPv6 OACK packet and returns the number of options that were found and optionally a list of pointers to the options in the packet.

If one or more of the option fields are not valid, then EFI_PROTOCOL_ERROR is returned and *OptionCount and *OptionList stop at the last valid option.

Status Codes Returned

<table>
<thead>
<tr>
<th>Code</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>EFI_SUCCESS</td>
<td>The OACK packet was valid and the OptionCount and OptionList parameters have been updated.</td>
</tr>
<tr>
<td>EFI_INVALID_PARAMETER</td>
<td>One or more of the following conditions is TRUE:</td>
</tr>
<tr>
<td></td>
<td>- PacketLen is 0.</td>
</tr>
<tr>
<td></td>
<td>- Packet is NULL or Packet is not a valid MTFTPv6 packet.</td>
</tr>
<tr>
<td></td>
<td>- OptionCount is NULL.</td>
</tr>
<tr>
<td>EFI_NOT_FOUND</td>
<td>No options were found in the OACK packet.</td>
</tr>
<tr>
<td>EFI_OUT_OF_RESOURCES</td>
<td>Storage for the OptionList array can not be allocated.</td>
</tr>
<tr>
<td>EFI_PROTOCOL_ERROR</td>
<td>One or more of the option fields is invalid.</td>
</tr>
</tbody>
</table>
EFI_MTFTP6_PROTOCOL.ReadFile()

Summary
Download a file from an MTFTPv6 server.

Prototype

typedef
  EFI_STATUS
(EFIAPI *EFI_MTFTP6_READ_FILE)(
    IN EFI_MTFTP6_PROTOCOL     *This,
    IN EFI_MTFTP6_TOKEN      *Token
  );

Parameters
This          Pointer to the EFI_MTFTP6_PROTOCOL instance.
Token         Pointer to the token structure to provide the parameters that
              are used in this operation. Type EFI_MTFTP6_TOKEN is defined
              in "Related Definitions" below.

Description
The ReadFile() function is used to initialize and start an MTFTPv6 download process and optionally
wait for completion. When the download operation completes, whether successfully or not, the
Token.Status field is updated by the EFI MTFTPv6 Protocol driver and then Token.Event is signaled if
it is not NULL.

Data can be downloaded from the MTFTPv6 server into either of the following locations:

- A fixed buffer that is pointed to by Token.Buffer
- A download service function that is pointed to by Token.CheckPacket

If both Token.Buffer and Token.CheckPacket are used, then Token.CheckPacket will be called
first. If the call is successful, the packet will be stored in Token.Buffer.

Related Definitions

//***************************************************************
// EFI_MTFTP6_TOKEN
//***************************************************************
typedef struct {
  EFI_STATUS           Status;
  EFI_EVENT            Event;
  EFI_MTFTP6_OVERRIDE_DATA OverrideData;
  UINT8                *Filename;
  UINT8                *ModeStr;
  UINT32               OptionCount;
  EFI_MTFTP6_OPTION*   OptionList;
  UINT64               BufferSize;
  VOID                 *Buffer;
} EFI_MTFTP6_TOKEN;
VOID     *Context;
EFI_MTFTP6_CHECK_PACKET CheckPacket;
EFI_MTFTP6_TIMEOUT_CALLBACK TimeoutCallback;
EFI_MTFTP6_PACKET_NEEDED PacketNeeded;
} EFI_MTFTP6_TOKEN;

status The status that is returned to the caller at the end of the
operation to indicate whether this operation completed
successfully. Defined Status values are listed below.

Event The event that will be signaled when the operation completes.
If set to NULL, the corresponding function will wait until the
read or write operation finishes. The type of Event must be
EVT_NOTIFY_SIGNAL.

OverrideData If not NULL, the data that will be used to override the existing
configure data. Type EFI_MTFTP6_OVERRIDE_DATA is defined in
EFI_MTFTP6_PROTOCOL.GetInfo().

Filename Pointer to the null-terminated ASCII file name string.

ModeStr Pointer to the null-terminated ASCII mode string. If NULL,
octet is used.

OptionCount Number of option/value string pairs.

OptionList Pointer to an array of option/value string pairs. Ignored if
OptionCount is zero. Both a remote server and this driver
implementation should support these options. If one or more
options are unrecognized by this implementation, it is sent to
the remote server without being changed. Type
EFI_MTFTP6_OPTION is defined in
EFI_MTFTP6_PROTOCOL.GetInfo().

BufferSize On input, the size, in bytes, of Buffer. On output, the number
of bytes transferred.

Buffer Pointer to the data buffer. Data that is downloaded from the
MTFTPv6 server is stored here. Data that is uploaded to the
MTFTPv6 server is read from here. Ignored if BufferSize is
zero.

Context Pointer to the context that will be used by CheckPacket,
TimeoutCallback and PacketNeeded.

CheckPacket Pointer to the callback function to check the contents of the
received packet. Type EFI_MTFTP6_CHECK_PACKET is defined
below.

TimeoutCallback Pointer to the function to be called when a timeout occurs.
Type EFI_MTFTP6_TIMEOUT_CALLBACK is defined below.

PacketNeeded Pointer to the function to provide the needed packet
contents. Only used in WriteFile() operation. Type
EFI_MTFTP6_PACKET_NEEDED is defined below.

The EFI_MTFTP6_TOKEN structure is used for both the MTFTPv6 reading and writing operations.
The caller uses this structure to pass parameters and indicate the operation context. After the reading or writing operation completes, the EFI MTFTPv6 Protocol driver updates the Status parameter and the Event is signaled if it is not NULL. The following table lists the status codes that are returned in the Status parameter.

### Status Codes Returned in the Status Parameter

<table>
<thead>
<tr>
<th>Status Code</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>EFI_SUCCESS</td>
<td>The data file has been transferred successfully.</td>
</tr>
<tr>
<td>EFI_OUT_OF_RESOURCES</td>
<td>Required system resources could not be allocated.</td>
</tr>
<tr>
<td>EFI_BUFFER_TOO_SMALL</td>
<td>BufferSize is not zero but not large enough to hold the downloaded data in downloading process.</td>
</tr>
<tr>
<td>EFI_ABORTED</td>
<td>Current operation is aborted by user.</td>
</tr>
<tr>
<td>EFI_NETWORK_UNREACHABLE</td>
<td>An ICMP network unreachable error packet was received.</td>
</tr>
<tr>
<td>EFI_NETWORK_UNREACHABLE2</td>
<td>An ICMP host unreachable error packet was received.</td>
</tr>
<tr>
<td>EFI_NETWORK_UNREACHABLE3</td>
<td>An ICMP protocol unreachable error packet was received.</td>
</tr>
<tr>
<td>EFI_NETWORK_UNREACHABLE4</td>
<td>An ICMP port unreachable error packet was received.</td>
</tr>
<tr>
<td>EFI_ICMP_ERROR</td>
<td>Some other ICMP ERROR packet was received.</td>
</tr>
<tr>
<td>EFI_TIMEOUT</td>
<td>No responses were received from the MTFTPv6 server.</td>
</tr>
<tr>
<td>EFI_TFTP_ERROR</td>
<td>An MTFTPv6 ERROR packet was received.</td>
</tr>
<tr>
<td>EFI_DEVICE_ERROR</td>
<td>An unexpected network error or system error occurred.</td>
</tr>
</tbody>
</table>

```
typedef EFI_STATUS (EFIAPI *EFI_MTFTP6_CHECK_PACKET)(
    IN EFI_MTFTP6_PROTOCOL *This,
    IN EFI_MTFTP6_TOKEN *Token,
    IN UINT16 PacketLen,
    IN EFI_MTFTP6_PACKET *Packet
);
```

- **This**: Pointer to the EFI_MTFTP6_PROTOCOL instance.
- **Token**: The token that the caller provided in the EFI_MTFTP6_PROTOCOL.ReadFile(), WriteFile() or ReadDirectory() function. Type EFI_MTFTP6_TOKEN is defined in EFI_MTFTP6_PROTOCOL.ReadFile().
- **PacketLen**: Indicates the length of the packet.
- **Packet**: Pointer to an MTFTPv6 packet. Type EFI_MTFTP6_PACKET is defined in EFI_MTFTP6_PROTOCOL.GetInfo().
EFI_MTFTP6_CHECK_PACKET is a callback function that is provided by the caller to intercept the EFI_MTFTP6_OPCODE_DATA or EFI_MTFTP6_OPCODE_DATA8 packets processed in the EFI_MTFTP6_PROTOCOL.ReadFile() function, and alternatively to intercept EFI_MTFTP6_OPCODE_OACK or EFI_MTFTP6_OPCODE_ERROR packets during a call to EFI_MTFTP6_PROTOCOL.ReadFile(), WriteFile() or ReadDirectory(). Whenever an MTFTPv6 packet with the type described above is received from a server, the EFI MTFTPv6 Protocol driver will call EFI_MTFTP6_CHECK_PACKET function to let the caller have an opportunity to process this packet. Any status code other than EFI_SUCCESS that is returned from this function will abort the transfer process.

```c
typedef
EFI_STATUS
(EFIAPI *EFI_MTFTP6_TIMEOUT_CALLBACK)(
    IN EFI_MTFTP6_PROTOCOL  *This,
    IN EFI_MTFTP6_TOKEN   *Token
);
```

This Pointer to the EFI_MTFTP6_PROTOCOL instance.
Token The token that is provided in the EFI_MTFTP6_PROTOCOL.ReadFile() or EFI_MTFTP6_PROTOCOL.WriteFile() or EFI_MTFTP6_PROTOCOL.ReadDirectory() functions by the caller. Type EFI_MTFTP6_TOKEN is defined in EFI_MTFTP6_PROTOCOL.ReadFile().

EFI_MTFTP6_TIMEOUT_CALLBACK is a callback function that the caller provides to capture the timeout event in the EFI_MTFTP6_PROTOCOL.ReadFile(), EFI_MTFTP6_PROTOCOL.WriteFile() or EFI_MTFTP6_PROTOCOL.ReadDirectory() functions. Whenever a timeout occurs, the EFI MTFTPv6 Protocol driver will call the EFI_MTFTP6_TIMEOUT_CALLBACK function to notify the caller of the timeout event. Any status code other than EFI_SUCCESS that is returned from this function will abort the current download process.
typedef EFI_STATUS
(EIFIAPI *EFI_MTFTP6_PACKET_NEEDED)(
    IN EFI_MTFTP6_PROTOCOL  *This,
    IN EFI_MTFTP6_TOKEN     Token,
    IN OUT UINT16           *Length,
    OUT VOID                **Buffer
);

This Pointer to the EFI_MTFTP6_PROTOCOL instance.
Token The token provided in the EFI_MTFTP6_PROTOCOL.WriteFile() by the caller.
Length Indicates the length of the raw data wanted on input, and the length the data available on output.
Buffer Pointer to the buffer where the data is stored.

EFI_MTFTP6_PACKET_NEEDED is a callback function that the caller provides to feed data to the EFI_MTFTP6_PROTOCOL.WriteFile() function. EFI_MTFTP6_PACKET_NEEDED provides another mechanism for the caller to provide data to upload other than a static buffer. The EFI MTFTP6 Protocol driver always calls EFI_MTFTP6_PACKET_NEEDED to get packet data from the caller if no static buffer was given in the initial call to EFI_MTFTP6_PROTOCOL.WriteFile() function. Setting *Length to zero signals the end of the session. Returning a status code other than EFI_SUCCESS aborts the session.

Status Codes Returned

<table>
<thead>
<tr>
<th>Code</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>EFI_SUCCESS</td>
<td>The data file is being downloaded.</td>
</tr>
<tr>
<td>EFI_INVALID_PARAMETER</td>
<td>One or more of the parameters is not valid.</td>
</tr>
<tr>
<td></td>
<td>• This is NULL.</td>
</tr>
<tr>
<td></td>
<td>• Token is NULL.</td>
</tr>
<tr>
<td></td>
<td>• Token.Filename is NULL.</td>
</tr>
<tr>
<td></td>
<td>• Token.OptionCount is not zero and</td>
</tr>
<tr>
<td></td>
<td>• Token.OptionList is NULL.</td>
</tr>
<tr>
<td></td>
<td>• One or more options in Token.OptionList have wrong</td>
</tr>
<tr>
<td></td>
<td>format.</td>
</tr>
<tr>
<td></td>
<td>• Token.Buffer and Token.CheckPacket are both NULL.</td>
</tr>
<tr>
<td></td>
<td>• Token.OverrideData.ServerIp is not a valid unicast</td>
</tr>
<tr>
<td></td>
<td>IPv6 address and not set to all zero.</td>
</tr>
<tr>
<td>EFI_UNSUPPORTED</td>
<td>One or more options in the Token.OptionList are not supported by this implementation.</td>
</tr>
<tr>
<td>EFI_NOT_STARTED</td>
<td>The EFI MTFTPv6 Protocol driver has not been started.</td>
</tr>
<tr>
<td>EFI_NO_MAPPING</td>
<td>The underlying IPv6 driver was responsible for choosing a source address for this instance, but no source address was available for use.</td>
</tr>
</tbody>
</table>
EFI_MTFTP6_PROTOCOL.WriteFile()

Summary
Send a file to an MTFTPv6 server. May be unsupported in some implementations.

Prototype
```c
typedef
EFI_STATUS
(EFI_API *EFI_MTFTP6_WRITE_FILE)(
    IN EFI_MTFTP6_PROTOCOL *This,
    IN EFI_MTFTP6_TOKEN   *Token
);
```

Parameters
- **This** Pointer to the EFI_MTFTP6_PROTOCOL instance.
- **Token** Pointer to the token structure to provide the parameters that are used in this function. Type EFI_MTFTP6_TOKEN is defined in EFI_MTFTP6_PROTOCOL.ReadFile().

Description
The WriteFile() function is used to initialize an uploading operation with the given option list and optionally wait for completion. If one or more of the options is not supported by the server, the unsupported options are ignored and a standard TFTP process starts instead. When the upload process completes, whether successfully or not, Token.Event is signaled, and the EFI MTFTPv6 Protocol driver updates Token.Status.

The caller can supply the data to be uploaded in the following two modes:
- Through the user-provided buffer
- Through a callback function

With the user-provided buffer, the Token.BufferSize field indicates the length of the buffer, and the driver will upload the data in the buffer. With an EFI_MTFTP6_PACKET_NEEDED callback function, the driver will call this callback function to get more data from the user to upload. See the definition of EFI_MTFTP6_PACKET_NEEDED for more information. These two modes cannot be used at the same time. The callback function will be ignored if the user provides the buffer.
Status Codes Returned

<table>
<thead>
<tr>
<th>Code</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>EFI_SUCCESS</td>
<td>The upload session has started.</td>
</tr>
<tr>
<td>EFI_UNSUPPORTED</td>
<td>The operation is not supported by this implementation.</td>
</tr>
<tr>
<td>EFI_INVALID_PARAMETER</td>
<td>One or more of the following conditions is \textit{TRUE}:</td>
</tr>
<tr>
<td></td>
<td>• This is \textit{NULL}.</td>
</tr>
<tr>
<td></td>
<td>• Token is \textit{NULL}.</td>
</tr>
<tr>
<td></td>
<td>• Token.Filename is \textit{NULL}.</td>
</tr>
<tr>
<td></td>
<td>• Token.OptionCount is not zero and</td>
</tr>
<tr>
<td></td>
<td>• Token.OptionList is \textit{NULL}.</td>
</tr>
<tr>
<td></td>
<td>• One or more options in \textit{Token.OptionList} have wrong</td>
</tr>
<tr>
<td></td>
<td>• format.</td>
</tr>
<tr>
<td></td>
<td>• Token.Buffer and Token.PacketNeeded are both \textit{NULL}</td>
</tr>
<tr>
<td></td>
<td>• Token.OverrideData.ServerIp is not a valid unicast IPv6</td>
</tr>
<tr>
<td></td>
<td>address and not set to all zero.</td>
</tr>
<tr>
<td>EFI_UNSUPPORTED</td>
<td>One or more options in the \textit{Token.OptionList} are not</td>
</tr>
<tr>
<td></td>
<td>supported by this implementation.</td>
</tr>
<tr>
<td>EFI_NOT_STARTED</td>
<td>The EFI MTFTPv6 Protocol driver has not been started.</td>
</tr>
<tr>
<td>EFI_NO_MAPPING</td>
<td>The underlying IPv6 driver was responsible for choosing a</td>
</tr>
<tr>
<td></td>
<td>source address for this instance, but no source address was</td>
</tr>
<tr>
<td></td>
<td>available for use.</td>
</tr>
<tr>
<td>EFI_ALREADY_STARTED</td>
<td>This \textit{Token} is already being used in another MTFTPv6</td>
</tr>
<tr>
<td></td>
<td>session.</td>
</tr>
<tr>
<td>EFI_OUT_OF_RESOURCES</td>
<td>Required system resources could not be allocated.</td>
</tr>
<tr>
<td>EFI_ACCESS_DENIED</td>
<td>The previous operation has not completed yet.</td>
</tr>
<tr>
<td>EFI_DEVICE_ERROR</td>
<td>An unexpected network error or system error occurred.</td>
</tr>
<tr>
<td>EFI_NO_MEDIA</td>
<td>There was a media error.</td>
</tr>
</tbody>
</table>

\texttt{EFI_MTFTP6_PROTOCOL.ReadDirectory()}

Summary
Download a data file directory from an MTFTPv6 server. May be unsupported in some implementations.

Prototype

\begin{verbatim}
typedef
EFI_STATUS
(EIFIAPI *EFI_MTFTP6_READ_DIRECTORY)(
    IN EFI_MTFTP6_PROTOCOL *This,
    IN EFI_MTFTP6_TOKEN       *Token
);
\end{verbatim}

Parameters
\begin{itemize}
    \item \texttt{This} Pointer to the \texttt{EFI_MTFTP6_PROTOCOL} instance.
\end{itemize}
Token

Pointer to the token structure to provide the parameters that are used in this function. Type `EFI_MTFTP6_TOKEN` is defined in `EFI_MTFTP6_PROTOCOL.ReadFile()`.

Description

The `ReadDirectory()` function is used to return a list of files on the MTFTPv6 server that are logically (or operationally) related to `Token.Filename`. The directory request packet that is sent to the server is built with the option list that was provided by caller, if present.

The file information that the server returns is put into either of the following locations:

- A fixed buffer that is pointed to by `Token.Buffer`
- A download service function that is pointed to by `Token.CheckPacket`

If both `Token.Buffer` and `Token.CheckPacket` are used, then `Token.CheckPacket` will be called first. If the call is successful, the packet will be stored in `Token.Buffer`.

The returned directory listing in the `Token.Buffer` or `EFI_MTFTP6_PACKET` consists of a list of two or three variable-length ASCII strings, each terminated by a null character, for each file in the directory. If the multicast option is involved, the first field of each directory entry is the static multicast IP address and UDP port number that is associated with the file name. The format of the field is `ip:ip:ip:port`. If the multicast option is not involved, this field and its terminating null character are not present.

The next field of each directory entry is the file name and the last field is the file information string. The information string contains the file size and the create/modify timestamp. The format of the information string is `filesize yyyy-mm-dd hh:mm:ss:ffff`. The timestamp is Coordinated Universal Time (UTC; also known as Greenwich Mean Time [GMT]).
### Status Codes Returned

<table>
<thead>
<tr>
<th>Code</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>EFI_SUCCESS</td>
<td>The MTFTPv6 related file &quot;directory&quot; has been downloaded.</td>
</tr>
<tr>
<td>EFI_UNSUPPORTED</td>
<td>The EFI MTFTPv6 Protocol driver does not support this function.</td>
</tr>
<tr>
<td>EFI_INVALID_PARAMETER</td>
<td>One or more of these conditions is <strong>TRUE</strong>:</td>
</tr>
<tr>
<td></td>
<td>• This is <strong>NULL</strong>.</td>
</tr>
<tr>
<td></td>
<td>• Token is <strong>NULL</strong>.</td>
</tr>
<tr>
<td></td>
<td>• Token.Filename is <strong>NULL</strong>.</td>
</tr>
<tr>
<td></td>
<td>• Token.OptionCount is not zero and Token.OptionList is <strong>NULL</strong>.</td>
</tr>
<tr>
<td></td>
<td>• One or more options in Token.OptionList have wrong format.</td>
</tr>
<tr>
<td></td>
<td>• Token.Buffer and Token.CheckPacket are both <strong>NULL</strong>.</td>
</tr>
<tr>
<td></td>
<td>• Token.OverrideData.ServerIp is not a valid unicast IPv6 address and not set to all zero.</td>
</tr>
<tr>
<td>EFI_UNSUPPORTED</td>
<td>One or more options in the Token.OptionList are not supported by this implementation.</td>
</tr>
<tr>
<td>EFI_NOT_STARTED</td>
<td>The EFI MTFTPv6 Protocol driver has not been started.</td>
</tr>
<tr>
<td>EFI_NO_MAPPING</td>
<td>The underlying IPv6 driver was responsible for choosing a source address for this instance, but no source address was available for use.</td>
</tr>
<tr>
<td>EFI_ALREADY_STARTED</td>
<td>This Token is already being used in another MTFTPv6 session.</td>
</tr>
<tr>
<td>EFI_OUT_OF_RESOURCES</td>
<td>Required system resources could not be allocated.</td>
</tr>
<tr>
<td>EFI_ACCESS_DENIED</td>
<td>The previous operation has not completed yet.</td>
</tr>
<tr>
<td>EFI_DEVICE_ERROR</td>
<td>An unexpected network error or system error occurred.</td>
</tr>
</tbody>
</table>

### EFI_MTFTP6_PROTOCOL.Poll()

**Summary**

Polls for incoming data packets and processes outgoing data packets.

**Prototype**

```c
typedef
EFI_STATUS
(EIFIAPI *EFI_MTFTP6_POLL) (  
    IN EFI_MTFTP6_PROTOCOL *This
);
```

**Parameters**

*This*

Pointer to the EFI_MTFTP6_PROTOCOL instance.

**Description**

The Poll() function can be used by network drivers and applications to increase the rate that data packets are moved between the communications device and the transmit and receive queues.
In some systems, the periodic timer event in the managed network driver may not poll the underlying communications device fast enough to transmit and/or receive all data packets without missing incoming packets or dropping outgoing packets. Drivers and applications that are experiencing packet loss should try calling the `Poll()` function more often.

**Status Codes Returned**

<table>
<thead>
<tr>
<th>Code</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>EFI_SUCCESS</td>
<td>Incoming or outgoing data was processed.</td>
</tr>
<tr>
<td>EFI_NOT_STARTED</td>
<td>This EFI MTFTPv6 Protocol instance has not been started.</td>
</tr>
<tr>
<td>EFI_INVALID_PARAMETER</td>
<td>This is <strong>NULL</strong>.</td>
</tr>
<tr>
<td>EFI_DEVICE_ERROR</td>
<td>An unexpected system or network error occurred.</td>
</tr>
<tr>
<td>EFI_TIMEOUT</td>
<td>Data was dropped out of the transmit and/or receive queue. Consider increasing the polling rate.</td>
</tr>
</tbody>
</table>
31 - EFI Redfish Service Support

31.1 EFI Redfish Discover Protocol

31.1.1 Overview

The purpose of the EFI Redfish Discover is to provide a mechanism for EFI Redfish clients to acquire the DMTF Redfish® services provided on the platform or network. See the Redfish Developer Hub at https://redfish.dmtf.org/ for official Redfish schema and specifications.

Redfish services can be discovered according to Redfish Host Interface (SMBIOS type 42) reported on platform, or optionally using Simple Service Discovery Protocol (SSDP) message over UDP port 1900 to search Redfish services which were joined well-known multicast group addresses. EFI Redfish Discover driver discovers Redfish services and creates EFI REST EX protocol instance for each Redfish service it found. It also configures EFI REST EX protocol instance according to the Redfish service information described in Redfish Host Interface or the response of UPnP M-SEARCH request (defined in UPnP Device Architecture, which can be obtained at “Links to UEFI-Related Documents” http://uefi.org/uefi).

EFI Redfish Discover Protocol behaves as a middle protocol which abstracts the creation and configuration of EFI REST EX instance from EFI Redfish clients.

- EFI Redfish Discover Protocol uses EFI UDP protocol to send SSDP message to verify or discover Redfish services. For the Redfish service reported by SMBIOS type 42h, EFI Redfish Discover Protocol can optionally unicast M-SEARCH request to Redfish service in order to verify the existence of service.

- EFI Redfish Discover Protocol can optionally provide the functionality of discovering Redfish services through each network interface installed on platform. Prior to acquiring the list of ready-to-use EFI REST EX protocol instances, the consumer of this protocol can get the network interface list and decide which interface is used for the multicast transmission. EFI Redfish Discover Protocol multicasts M-SEARCH request to multicast group addresses then collects M-SEARCH responses from Redfish services in asynchronous or synchronous manner.

- EFI Redfish Discover Protocol provides the information of each network interface installed on platform through GetNetworkInterfaceList() function. The information such as MAC address, subnet ID, subnet mask and VLAN ID of network interface could be utilized by upper-layer EFI application or driver to identify network interface used for Redfish service discovery. EFI Redfish Discover Protocol abstracts EFI network stack to user which means this protocol should not require user to configure UDP before utilizing services. Network configuration of network interface such as station IP address, subnet ID, subnet mask and other operational parameters should be configured through system firmware specific implementation (for example system utility). This protocol should simply use UDP default station properties.

Multicast across internetworks is handled by multicast router and is not in the scope of EFI Redfish Discover Protocol. The implementation of upper-layer user interface is system firmware design-specific.

- EFI Redfish Discovery Protocol is the helper driver to discover Redfish services on platform or network. The upper level EFI Redfish client could provide its own implementation of how to utilize information returned from this protocol. Such as network interface selection UI, create...
Redfish host interface (SMBIOS type 42h) according to Redfish services information, configure system BIOS setting using Redfish service or etc.

31.1.2 EFI Redfish Discover Driver

A Redfish Discover Driver installs the Redfish Discover Protocol and EFI Driver Binding Protocol in its driver entry point.

The Driver Binding Protocol contains three services. These are `Supported()`, `Start()`, and `Stop()`.

- `Supported()` tests to see if the Redfish Discover Driver can manage a device handle. A Redfish Discover Driver can manage device handle that contain the EFI REST EX Service Binding Protocol, EFI UDP4 Service Binding Protocol or EFI UDP6 Service Binding Protocol, so a Redfish Discover Driver must look for these three protocols on the device handle that is being tested, and return success if any of them is presented.

- The `Start()` function tells the Redfish Discover Driver to start managing a device driver. The device handle should support at least one of the service binding protocols checked in `Supported()`. The Redfish Discover Driver should create a child handle for each service binding protocol, and open these children with `BY_DRIVER` attribute.

- The `Stop()` function tells the Redfish Discover Driver to stop managing a device driver. The `Stop()` function can destroy one or more of the device handles (or its child handles) that being managed by Redfish Discover Driver. A Redfish Discover Driver should stop the in-process discovery and destroy corresponding child handle which was created in a previous call to `Start()`, or in `AcquireRedfishService()`.
31.1.3 EFI Redfish Discover Client

An EFI Redfish client invokes EFI Redfish Discover Protocol to acquire the ready-to-use EFI REST EX protocol instance.

Below is the conceptual figure of mechanism of EFI Redfish Discover Protocol. The first scenario is unicast M-SEARCH to verify Redfish service reported in SMBIOS type 42h.

1. EFI Redfish client invokes EFI Redfish Discover Protocol to acquire ready-to-use EFI REST EX for communicating with Redfish services reported in Redfish Host Interface (SMBIOS type 42h).
2. EFI Redfish Discover Protocol optionally verifies the existence of Redfish service by unicasting M-SEARCH to Redfish service according to the Redfish service information provided in Redfish Host Interface.
3. EFI Redfish Discover Protocol creates and configures REST EX instance for Redfish service according to the Redfish service information provided in Redfish Host Interface.
4. EFI Redfish clients communicate with Redfish service using EFI REST EX instance returned from EFI Redfish Discover protocol.

EFI Redfish client passes EFI_REDFISH_DISCOVERED_TOKEN and the discovery options to EFI Redfish Discover Protocol. EFI_EVENT is created by EFI Redfish client for retrieving EFI_REDFISH_DISCOVERED_LIST once EFI Redfish Discover Protocol optionally verifies Redfish service reported by Redfish Host Interface. EFI Redfish client can listen to the notification of verified...
Redfish service in asynchronous or synchronous according to the setting of options indicated in `EFI_REDFISH_DISCOVER_FLAG`.

The second scenario is optionally provided by EFI Redfish Discover Protocol, which is multicast M-SEARCH to discover Redfish services.

1. EFI Redfish client gets the list of network interfaces if it would like to discover Redfish services on the certain network.
2. EFI Redfish client invokes EFI Redfish Discover Protocol to acquire ready-to-use EFI REST EX for communicating with Redfish services.
3. EFI Redfish Discover Protocol discovers Redfish services through SSDP over UDP.
4. EFI Redfish clients communicate with Redfish service using EFI REST EX instance returned from EFI Redfish Discover protocol.

EFI Redfish client passes `EFI_REDFISH_DISCOVERED_TOKEN` and the discovery options to EFI Redfish Discover Protocol. `EFI_EVENT` is created by EFI Redfish client for retrieving `EFI_REDFISH_DISCOVERED_LIST` when any time EFI Redfish Discover Protocol discovers new Redfish service. EFI Redfish client can listen to the notification of new found Redfish service in asynchronous or synchronous according to the setting of options indicated in `EFI_REDFISH_DISCOVER_FLAG`. Setting `Timeout` to zero in `EFI_REDFISH_DISCOVERED_TOKEN` to waiting for the new discovered Redfish service in synchronously, otherwise asynchronous notification happens when new Redfish service is discovered by EFI Redfish Discover Protocol.
31.1.4 EFI Redfish Discover Protocol

Summary
This protocol is utilized by EFI Redfish clients to acquire the list of Redfish services provided on platform or network.

Protocol GUID
#define EFI_REDFISH_DISCOVER_PROTOCOL_GUID \
{0x5db12509, 0x4550, 0x4347, \
 {0x96, 0xb3, 0x73, 0xc0, 0xff, 0x6e, 0x86, 0x9f}}

Protocol Interface Structure
typedef struct _EFI_REDFISH_DISCOVER_PROTOCOL {
    EFI_REDFISH_DISCOVER_NETWORK_LIST    GetNetworkInterfaceList;
    EFI_REDFISH_DISCOVER_ACQUIRE_SERVICE AcquireRedfishService;
    EFI_REDFISH_DISCOVER_ABORT_ACQUIRE   AbortAcquireRedfishService;
    EFI_REDFISH_DISCOVER_RELEASE_SERVICE ReleaseRedfishService;
} EFI_REDFISH_DISCOVER_PROTOCOL;

Parameters
    GetNetworkInterfaceList Get the list of network interfaces on which Redfish services could be discovered.
    AcquireRedfishService   Acquire the list of Redfish services.
    AbortAcquireRedfishService Abort Redfish services acquire process.
    ReleaseRedfishService   Release Redfish services acquired from AcquireRedfishService().

Description
EFI Redfish Discover Protocol provides a mechanism for EFI Redfish clients to acquire the Redfish services provided on the platform or network as described before.

EFI_REDFISH_DISCOVER_PROTOCOL.GetNetworkInterfaceList ()

Summary
Get the currently available list of network interfaces on which Redfish services could be discovered.
Protocol Interface

```c
typedef
EFI_STATUS
(EFIAPI  *EFI_REDFISH_DISCOVER_NETWORK_LIST)(
    IN     EFI_REDFISH_DISCOVER_PROTOCOL *This,
    IN     EFI_HANDLE              ImageHandle,
    OUT    UINTN                  *NumberOfNetworkInterfaces,
    OUT    EFI_REDFISH_DISCOVER_NETWORK_INTERFACE **NetworkInterfaces
);
```

Parameters

- **This**: This is the `EFI_REDFISH_DISCOVER_PROTOCOL` instance.
- **ImageHandle**: EFI image to get network list. The image handle is caller’s image handle.
- **NumberOfNetworkInterfaces**: Number of network interfaces in `NetworkInterfaces`.
- **NetworkInterfaces**: It's an array of instances. The number of entries in `NetworkInterfaces` is indicated by `NumberOfNetworkInterfaces`. Caller has to release the memory allocated by Redfish discover protocol with a call to `EFI_BOOT_SERVICES.FreePool()`.

Description

This function is used to get the list of network interfaces which can be used to send SSDP message over UDP protocol for the Redfish services discovery. The entry in `NetworkInterfaces` could be used as the parameter to `EFI_REDFISH_DISCOVER_PROTOCOL.AcquireRedfishService` function for discovering Redfish service on specific network interface.

Related Description

```c
//*******************************************************
// EFI_REDFISH_DISCOVER_NETWORK_INTERFACE
//*******************************************************
typedef struct {
    EFI_MAC_ADDRESS           MacAddress;
    BOOLEAN                   IsIpv6;
    EFI_IP_ADDRESS            SubnetId;
    UINT8                      SubnetPrefixLength;
    UINT16                     VlanId;
} EFI_REDFISH_DISCOVER_NETWORK_INTERFACE;
```

Parameters

- **MacAddress**: MAC address of this network interface.
- **IsIpv6**: If TRUE, indicates the network interface is running IPv6. Otherwise the network interface is running IPv4.
- **SubnetId**: Subnet of this network.
- **SubnetPrefixLength**: Subnet prefix-length for IPv4 and IPv6.
- **VlanId**: VLAN ID of this network interface.
Status Codes Returned

<table>
<thead>
<tr>
<th>Code</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>EFI_SUCCESS</td>
<td>Network interface is returned in NetworkInterfaces and the number of network interfaces is returned in NumberOfNetworkInterfaces successfully.</td>
</tr>
<tr>
<td>EFI_INVALID_PARAMETER</td>
<td>One of below parameters is NULL. ImageHandle, NumberOfNetworkInterfaces, and NetworkInterfaces</td>
</tr>
<tr>
<td>EFI_UNSUPPORTED</td>
<td>Unable to return network interface list.</td>
</tr>
<tr>
<td>EFI_NOT_FOUND</td>
<td>No network interfaces are found.</td>
</tr>
<tr>
<td>EFI_OUT_OF_RESOURCE</td>
<td>Not enough resources to return network interfaces to caller.</td>
</tr>
</tbody>
</table>

** EFI_REDFISH_DISCOVER_PROTOCOL.AcquireRedfishService ()

**Summary**
This function acquires the list of discovered Redfish services.

**Protocol Interface**

```c
typedef
EFI_STATUS
(EIFIAPI *EFI_REDFISH_DISCOVER_ACQUIRE_SERVICE)(
    INEFI_REDFISH_DISCOVER_PROTOCOL *This,
    INEFI_HANDLE ImageHandle,
    INEFI_REDFISH_DISCOVER_NETWORK_INTERFACE *TargetNetworkInterface
    OPTIONAL,
    INEFI_REDFISH_DISCOVER_FLAG Flags,
    INEFI_REDFISH_DISCOVERED_TOKEN *Token
)
```

**Parameters**

- **This**
  This is the EFI_REDFISH_DISCOVER_PROTOCOL instance.
- **ImageHandle**
  EFI image acquires Redfish service discovery. The image handle is caller’s image handle.
- **TargetNetworkInterface**
  The target Network Interface which is used to discover Redfish services. Set to NULL to discover Redfish services on all network interfaces.
- **Flags**
  Options of Redfish service discovery.
- **Token**
  EFI_REDFISH_DISCOVERED_TOKEN instance. The memory of EFI_REDFISH_DISCOVERED_LIST and the strings in EFI_REDFISH_DISCOVERED_INFORMATION are all allocated by AcquireRedfishService() and must be freed when caller invokes ReleaseRedfishService().

**Description**

This function is used to acquire the list of Redfish services which are discovered according to Redfish Host Interface or through SSDP over UDP. Redfish services discovery through SSDP over UDP could be
achieved via network interface specified in TargetNetworkInterface or via all network interfaces if TargetNetworkInterface is specified as NULL. **EFI_REDFISH_DISCOVERED_LIST** is returned to EFI Redfish client by signaling the EFI event created by client. Each of EFI handle in **EFI_REDFISH_DISCOVERED_LIST** has the corresponding EFI REST EX instance installed on it. Each REST EX instance is a child instance which is created through EFI REST EX service binding protocol and used by EFI Redfish client for communicating with specific Redfish service. In AcquireRedfishService(), UDP child is created and opened to do SSDP discovery. This UDP child will be destroyed right away after the discovery is done. AcquireRedfishService() also creates and opens REST EX child to configures REST EX instance according to Redfish service information retuned in M-SEARCH response or Redfish Host Interface. REST EX child must be closed after REST EX child is configured. EFI Redfish client must open REST EX instance from RedfishRestExHandle returned in **EFI_REDFISH_DISCOVERED_INFORMATION** and close REST EX instance once EFI Redfish client is no longer communicating with Redfish service.

**Related Description**

```
//***************************************************************
// EFI_REDFISH_DISCOVER_FLAG
//***************************************************************
#define EFI_REDFISH_DISCOVER_HOST_INTERFACE   0x00000001
#define EFI_REDFISH_DISCOVER_SSDP             0x00000002
#define EFI_REDFISH_DISCOVER_SSDP_UDP6        0x00000004
#define EFI_REDFISH_DISCOVER_KEEP_ALIVE       0x00000008
#define EFI_REDFISH_DISCOVER_RENEW            0x00000010
#define EFI_REDFISH_DISCOVER_VALIDATION       0x80000000
#define EFI_REDFISH_DISCOVER_DURATION_MASK    0x0f000000
```

**EFI_REDFISH_DISCOVER_FLAG** is used to indicate the options when EFI Redfish clients acquire Redfish discover list through this protocol. Redfish Discover Protocol discovers Redfish service according to Redfish Host Interface when **EFI_REDFISH_DISCOVER_HOST_INTERFACE** is set to **TRUE**. Redfish Discover Protocol also optionally discovers Redfish services using SSDP UPnP M-SEARCH request through UDP Port 1900. Redfish Discover Protocol returns **EFI_INVALID_PARAMETER** if none of **EFI_REDFISH_DISCOVER_HOST_INTERFACE** and **EFI_REDFISH_DISCOVER_SSDP** is set to **TRUE**. Set **EFI_REDFISH_DISCOVER_SSDP_UDP6** to indicate using IPv6 as internet protocol. For the Redfish service discovery according to Redfish Host Interface, Redfish service information like IP address is described in Redfish Host Interface. EFI Redfish client can set **EFI_REDFISH_DISCOVER_VALIDATION** to **TRUE** to ask Redfish Discover Protocol to validate this Redfish service using IP address described in Redfish Host Interface. Redfish Discover Protocol unicasts UPnP M-SEARCH request to the target Redfish service and verify the response message to determine if the target Redfish service is existing or not. **EFI_REDFISH_DISCOVER_VALIDATION** doesn’t affect the SSDP discovery. For Redfish SSDP discovery, the responses of the multicast UPnP M-SEARCH request imply the valid Redfish services are existing.

According to UPnP device architecture, the maximum waiting time of the response to UPnP M-SEARCH request is indicated in MX message header. The value is greater or equal to 1 to less than 5 inclusive in second. In order to give the chance to those Redfish services which do not respond to M-SEARCH in time, set **EFI_REDFISH_DISCOVER_KEEP_ALIVE** to **TRUE** to tell Redfish Discover Protocol keeps to sending multicast M-SEARCH request. The duration of periodical multicast request is declared in **EFI_REDFISH_DISCOVER_DURATION_MASK**. The value indicated in
**EFI_REDFISH_DISCOVER_DURATION_MASK** means 2 to the power of duration. The valid value of duration is greater or equal to 3 and less or equal to 15. The corresponding duration is 8 to $2^{15}$ seconds. Minimum duration is set to 8 seconds in order to keep the duration out of scope of MX value defined in UPnP device architecture. Duration is only valid when **EFI_REDFISH_DISCOVER_KEEP_ALIVE** is set to TRUE and **EFI_REDFISH_DISCOVER_SSDP** is set to TRUE.

Redfish Discover Protocol maintains an internal database of Redfish services it found. It also maintains the EFI image which owns the EFI REST EX instance of discovered Redfish services. Redfish Discover Protocol only signals EFI Redfish client with new found of Redfish services instead of notifying EFI Redfish client the duplicate Redfish services found earlier, unless **EFI_REDFISH_DISCOVER_RENEW** is set to TRUE. Set **EFI_REDFISH_DISCOVER_RENEW** to TRUE forces Redfish Discover Protocol to notify EFI Redfish clients all found Redfish services, even the Redfish service which was already discovered and notified previously.

```c
//******************************************************************************
//  EFI_REDFISH_DISCOVERED_TOKEN
//******************************************************************************
#define REDFISH_DISCOVER_TOKEN_SIGNATURE SIGNATURE_32 ('R', 'F', 'T', 'S')

typedef struct {
    UINT32 Signature;
    EFI_REDFISH_DISCOVERED_LIST DiscoveredList;
    EFI_EVENT Event;
    UINTN Timeout;
} EFI_REDFISH_DISCOVERED_TOKEN;

Description

**EFI_REDFISH_DISCOVERED_TOKEN** is created by EFI Redfish client and passed to AcquireRedfishService().

Parameters

- **Signature**
  - The token signature should be the value of REDFISH_DISCOVER_TOKEN_SIGNATURE defined above.

- **DiscoveredList**
  - Structure of **EFI_REDFISH_DISCOVERED_LIST** to retrieve the discovered Redfish services.

- **Event**
  - EFI event at the TPL_CALLBACK level created by EFI Redfish client, which is used to be notified when Redfish services are discovered or any errors occurred during discovery.

- **Timeout**
  - The timeout value declared in **EFI_REDFISH_DISCOVERED_TOKEN** determines the seconds to drop discovery process. Basically, the nearby Redfish services must give the response in $>=1$ and $<=5$ seconds. The valid timeout value used for the asynchronous discovery is $>=1$ and $<=5$ seconds. Set the timeout to zero means to discover Redfish service synchronously.
//*******************************************************************************
// EFI_REDFISH_DISCOVERED_LIST
//*******************************************************************************
typedef struct {
    UINTN NumberOfServiceFound;
    EFI_REDFISH_DISCOVERED_INSTANCE *RedfishInstances;
} EFI_REDFISH_DISCOVERED_LIST;

Description
The content of **EFI_REDFISH_DISCOVERED_LIST** is filled by *AcquireRedfishService()* before signaling Event. NumberOfServiceFound must be set to 0 and RedfishInstances must be NULL when client invokes *AcquireRedfishService()* . The memory block for RedfishInstances is allocated by the EFI Redfish Discover Protocol, and will be freed by the EFI Redfish Discover Protocol as well in *ReleaseRedfishService()*.

Parameters
- **NumberOfServiceFound**: Number of Redfish services are discovered.
- **RedfishInstances**: Pointer to **EFI_REDFISH_DISCOVERED_INSTANCE**, number of Redfish services are discovered is indicated in NumberOfServiceFound.

//*******************************************************************************
// EFI_REDFISH_DISCOVERED_INSTANCE
//*******************************************************************************
typedef struct {
    EFI_STATUS Status;
    EFI_REDFISH_DISCOVERED_INFORMATION Information;
} EFI_REDFISH_DISCOVERED_INSTANCE;

Description
This structure describes the status and the information of discovered Redfish service.

Parameters
- **Status**: EFI status code of Redfish service discovery.
- **Information**: The information of Redfish service discovered. The information is only valid when Status is EFI_SUCCESS. Refer to below description of **EFI_REDFISH_DISCOVERED_INSTANCE**.
typedef struct {
    EFI_HANDLE RedfishRestExHandle;
    BOOLEAN IsIPv6;
    EFI_IP_ADDRESS RedfishHostIpAddress;
    UINT16 RedfishVersion;
    CHAR16 *Location;
    CHAR16 *Uuid;
    CHAR16 *Os;
    CHAR16 *OsVersion;
    CHAR16 *Product;
    CHAR16 *ProductVersion;
    BOOLEAN UseHttps;
} EFI_REDFISH_DISCOVERED_INFORMATION;

Description
This structure describes each Redfish service information. The corresponding EFI REST EX protocol instance is also created and configured by EFI Redfish Discover Protocol for EFI Redfish client. The memory allocated for the information in this structure will be freed by EFI Redfish Discover Protocol in ReleaseRedfishService().

Parameters
RedfishRestExHandle EFI handle which has EFI REST EX protocol instance installed on it. The EFI REST EX protocol instance is already configured by EFI Redfish Discover Protocol through EFI_REST_EX_PROTOCOL.Configure() according to the Redfish host information discovered through Redfish Host Interface or SSDP.
IsIPv6 Indicates the Redfish service is reached via IPv6 protocol.
RedfishHostIpAddress Redfish service host IP address.
RedfishVersion Redfish service version. The high byte of RedfishVersion is the major Redfish service version, low byte is the minor Redfish version. For example 0x100 is Redfish service. Redfish service version is acquired from “ST” header in the response of M-SEARCH request.
Location Redfish service host location, this information is acquired from “Server” header returned in the response of M-SEARCH request.
Uuid The UUID of Redfish service, this information is acquired from “USN” header defined in UPnP Device Architecture specification.
Os The OS provides Redfish service, this information is acquired from “Server” header returned in the response of M-SEARCH request. Below is the response in “Server” header defined in UPnP Device Architecture specification.
    SERVER:OS/version UPnP/1.1 product/version
OsVersion Redfish service OS version, this information is acquired from “Server” header returned in the response of M-SEARCH request. Below is the
response in “Server” header defined in UPnP Device Architecture specification.
SERVER:OS/version UPnP/1.1 product/version

**Product**
Product name, this information is extracted from “Server” header returned in the response of M-SEARCH request. Below is the response in “Server” header defined in UPnP Architecture Device specification.
SERVER:OS/version UPnP/1.1 product/version

**ProductVersion**
Product version, this information is acquired from “Server” header returned in the response of M-SEARCH request. Below is the response in “Server” header defined in UPnP Device Architecture specification.
SERVER:OS/version UPnP/1.1 product/version

**UseHttps**
Indicates the Redfish service is reached via HTTPS protocol.

### Status Codes Returned

<table>
<thead>
<tr>
<th>Status Code</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>EFI_SUCCESS</td>
<td>Acquire for Redfish service list is successful.</td>
</tr>
<tr>
<td>EFI_INVALID_PARAMETER</td>
<td>One or more of the following is TRUE:</td>
</tr>
<tr>
<td>This is NULL.</td>
<td></td>
</tr>
<tr>
<td>ImageHandle is NULL.</td>
<td></td>
</tr>
<tr>
<td>Flags is 0 or the improper bit combination of option is set in Flag.</td>
<td></td>
</tr>
<tr>
<td>Token is NULL.</td>
<td></td>
</tr>
<tr>
<td>Token-&gt;Timeout is greater than 5 seconds.</td>
<td></td>
</tr>
<tr>
<td>Token-&gt;Event is NULL.</td>
<td></td>
</tr>
<tr>
<td>On input,</td>
<td></td>
</tr>
<tr>
<td>Token-&gt;DiscoveredList.NumberOfServiceFound is not 0,</td>
<td></td>
</tr>
<tr>
<td>or Token-&gt;DiscoveredList-&gt;RedfishInstances is not NULL.</td>
<td></td>
</tr>
<tr>
<td>Others</td>
<td>Fail to acquire the list of Redfish service.</td>
</tr>
</tbody>
</table>

### EFI_REDFISH_DISCOVER_PROTOCOL.AbortAcquireRedfishService()

**Summary**
This function aborts Redfish service discovery on the given network interface.

**Protocol Interface**

typedef
EFI_STATUS
(EIFIAPI *EFI_REDFISH_DISCOVER_ABORT_ACQUIRE)(
    IN EFI_REDFISH_DISCOVER_PROTOCOL *This,
    IN EFI_REDFISH_DISCOVER_NETWORK_INTERFACE *TargetNetworkInterface OPTIONAL);

**Parameters**

**This**
This is the EFI_REDFISH_DISCOVER_PROTOCOL instance.

**TargetNetworkInterface**
The target Network Interface on which Redfish services discovery is in process. NULL to abort Redfish service discovery on all network interfaces.
Description
In `AbortAcquireRedfishService()`, to abort the in-process Redfish service, discovery is required for preventing unexpected behaviors from happening. This function has to cancel in-process SSDP, the unicast over Udp4/Udp6, close Udp4/Udp6 protocol and destroy the Udp4/Udp6 child. Also closes REST EX opened for configuring REST EX child instance.

Status Codes Returned

<table>
<thead>
<tr>
<th>Code</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>EFI_SUCCESS</td>
<td>Redfish service discovery is aborted.</td>
</tr>
<tr>
<td>EFI_INVALID_PARAMETER</td>
<td>One or more of the following is TRUE:</td>
</tr>
<tr>
<td></td>
<td>- This is NULL.</td>
</tr>
</tbody>
</table>

**EFI_REDFISH_DISCOVER_PROTOCOL.ReleaseRedfishService ()**

Summary
This function releases the list of Redfish services discovered previously.

Protocol Interface

```c
typedef EFI_STATUS
  (EFIAPI *EFI_REDFISH_DISCOVER_RELEASE_SERVICE)(
    IN EFI_REDFISH_DISCOVER_PROTOCOL *This,
    IN EFI_REDFISH_DISCOVERED_LIST *List
  );
```

Parameters

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>This</td>
<td>This is the <code>EFI_REDFISH_DISCOVER_PROTOCOL</code> instance.</td>
</tr>
<tr>
<td>List</td>
<td>The pointer to <code>EFI_REDFISH_DISCOVERED_LIST</code> which lists the Redfish services to release.</td>
</tr>
</tbody>
</table>

Description
The Redfish services which listed in List will be released in `ReleaseRedfishService()`. All memory blocks which were allocated for Redfish service information will be freed in this function. EFI REST EX protocol instance which was created in `AcquireRedfishService()` will be also destroyed in `ReleaseRedfishService()`. The Redfish service listed in *List is not required to be identical or in the same order with `EFI_REDFISH_DISCOVERED_LIST` retuned from `AcquireRedfishService()`. List is flexible to list any Redfish services which were discovered by `AcquireRedfishService()` earlier. In `ReleaseRedfishService()`, free the resource allocated for the discovered Redfish service indicated in `EFI_REDFISH_DISCOVERED_LIST`. 
Status Codes Returned

<table>
<thead>
<tr>
<th>Status Code</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>EFI_SUCCESS</td>
<td>The Redfish services listed in <code>*List</code> are released successfully.</td>
</tr>
<tr>
<td>EFI_INVALID_PARAMETER</td>
<td>One or more of the following is TRUE:</td>
</tr>
<tr>
<td></td>
<td>- <code>This</code> is NULL.</td>
</tr>
<tr>
<td></td>
<td>- <code>List</code> is NULL.</td>
</tr>
<tr>
<td></td>
<td>- Invalid settings in <code>*List</code>.</td>
</tr>
</tbody>
</table>

31.1.5 Implementation Examples

31.1.5.1 Processes to Discover Redfish Services

The following flowchart delineates the EFI Redfish client processes of utilizing EFI Discover Protocol to discover Redfish service, abort discovery and release discovered Redfish service instance.

![Flowchart](image)

31.1.5.2 Network Interface Configuration

The EFI Redfish Discover Protocol provides a Redfish service discovery function to discover Redfish service through SMBIOS type 42 or optionally discover Redfish service on specific network interface. EFI Redfish Clients (EFI driver or EFI Application) can utilize the discover function to acquire Redfish service and manipulate Redfish properties to manage a system. For example, applying BIOS settings on the systems managed by Redfish Service. The system could be the one that runs EFI Redfish Client, or other systems on the network. If Redfish service is discovered according to SMBIOS type 42, then the platform developer has to create an SMBIOS type 42 entry with host (station) and Redfish Service information (Refer to DSP0270, Redfish Host Interface Specification). Besides discovering Redfish service using SMBIOS type 42, Redfish services can be also discovered by using SSDP over UDP. However, the network interface must be configured using either DHCP or static configuration prior to discovery of Redfish services. If the network interface is configured statically, then at least the IP address and Subnet mask must be configured for the station. The VLAN ID and new route entry may need to be configured depending on the networking environment if necessary.
Below is the implementation example for configuring network interface. Network interface could be configured in platform-implementation method. For example, platform developer can provide HII network options in BIOS setup utility. Network interface could be configured in statically or dynamically (DHCP) manner and the configuration could be stored in EFI variables or any platform non-volatile storage which may consumed by network stacks when each time system boot. This makes sure certain network interface is configured properly before EFI Redfish Clients utilizing EFI Redfish Discover Protocol.

The alternative of configuring network stack is system boots to EFI Shell and execute ifconfig shell command. This configures the settings of certain network interfaces. After this, network interface is ready to process Redfish service discovery by EFI Redfish Clients. However, this method requires user to configure network interface when each time system boot to EFI shell, unless other implementations of ifconfig EFI shell command is provided.

Once EFI Redfish Client is launched, it gets network interface information using EFI Redfish Discover protocol. EFI Redfish Client may provide selection UI of network interfaces for Redfish service discovery. EFI Redfish Client could manipulate Redfish properties such as BIOS Attributes on the discovered Redfish services for system management or deployment. EFI Redfish Client can also optionally maintain the information, location and other properties of discovered Redfish services in non-volatile storage for next system boots afterward.
31.2 EFI Redfish JSON Structure Converter

31.2.1 The Guidance of Writing EFI Redfish JSON Structure Converter

To provide interoperability between the Redfish service and the EFI environment, EFI Redfish JSON structure converters for each Redfish schema namespace should be implemented for EFI Redfish clients. This recommendation of writing EFI Redfish JSON structure converters is necessary to unify the implementation and capability of the converters.

- One converter supports one Redfish schema resource type; write the converter based on Redfish resource type. Using Redfish schema as an example:
  - AccountService.v1_0_0.json: RedfishAccountService_V1_0_0_Dxe driver
  - AttributeRegistry.v1_2_0.json: RedfishAttributeRegistry_V1_2_0_Dxe driver
  - EthernetInterface.v1_4_0.json: EthernetInterface_V1_4_0_Dxe driver

- Redfish JSON structure converter can be delivered in source code package or binary (library or EFI driver) format.

- A C header file must be released with the Redfish JSON structure converter package. The package could be provisioned to conform to any EFI implementation, such as EFI EDKII open source.

- Provide documents which can describe the usage of structure members defined in REST JSON structure.

- The documentation can be published with a source code package, binary package, web site, online help, etc.

- Write the converter as an EFI DXE driver, and utilize EFI_REST_JSON_STRUCTURE_PROTOCOL to register the converter to provide the corresponding EFI_REST_JSON_STRUCTURE_PROTOCOL functions:
  - ToStructure()
  -ToJson()
  -DestoryStructure()

**EFI_REST_JSON_RESOURCE_TYPE_IDENTIFIER**

**Namespace**

- **ResourceTypeName**: String to Redfish schema resource type.
- **MajorVersion**: String to Redfish schema major version, NULL string for non version controlled schema.
- **MinorVersion**: String to Redfish schema minor version, NULL string for non version controlled schema.
- **ErrataVersion**: String to Redfish schema errata version, NULL string for non version controlled schema.

**Datatype**

String to data type defined in Redfish schema

**Examples**

AccountService.v1_0_0.json
Namespace
   ResourceTypeName: "AccountService"
   MajorVersion: "1"
   MinorVersion: "0"
   ErrataVersion: "0"

Datatype: "AccountService"

Namespace
   ResourceTypeName: "ComputerSystemCollection"
   MajorVersion: NULL
   MinorVersion: NULL
   ErrataVersion: NULL

Datatype: "ComputerSystemCollection"

- Determine Redfish resource type according to the given JsonRsrcIdentifier. If the given JsonRsrcIdentifier is non-NULL, the Redfish resource structure converter must convert the JSON resource to the Redfish JSON structure according to the resource type and revision specified in JsonRsrcIdentifier. The converter should not refer to the resource type and revision according to Redfish namespace and datatype indicated in “odata.type” in JSON text resource. This prevents from the returned structure format is different with what consumer expects.

- Automatically determine the Redfish resource type. If the given JsonRsrcIdentifier is NULL, the EFI Redfish JSON structure converter should check the namespace and datatype indicated in “odata.type” in the JSON text resource. Parse this identifier property to retrieve the corresponding Redfish schema name space and data type, then decode the JSON text resource into the corresponding structure. EFI_REST_JSON_RESOURCE_TYPE_IDENTIFIER in JsonStructure returned to consumer should be filled with the correct Redfish schema resource type information following the guidance mentioned above.

- All structure members for Redfish schema must be declared as C pointers. With this, the converter consumer can get the partial Redfish JSON properties from the converter. The consumer just initializes certain structure members, and the converter producer only converts non-NULL pointers in the given structure into corresponding Redfish JSON properties in text format.

31.2.2 The Guidance of Using EFI Redfish JSON Structure Converter

The consumer of EFI Redfish JSON structure converter utilizes EFI_REST(JSON) STRUCTURE_PROTOCOL for converting Redfish JSON resource to Redfish JSON structure and vice versa.

Refer to the converter document to include the C header file of the Redfish JSON structure converter into the build process. For example, include the converter’s EDKII package into an EFI module INF file for the C header file reference, or follow the build rule of other EFI implementations.
There are two ways for a consumer to convert JSON resources using the
**EFI_REST_JSON_STRUCTURE_PROTOCOL**:

- **Setup the correct Redfish namespace and datatype in**
  **EFI_REST_JSON_RESOURCE_TYPE_IDENTIFIER**. This makes sure the EFI REST JSON Structure Protocol uses the exact converter that the consumer prefers for the conversion. In this case, the Redfish namespace and datatype indicated in “odata.type” in the **EFI_REST_JSON_RESOURCE_TYPE_IDENTIFIER** is set to NULL. This means the converter may recognize the Redfish namespace and datatype indicated in “odata.type” in the JSON text resource, and converts it to the C structure it supports. In this case, the consumer has to be careful when using a C structure pointer to refer to the Redfish JSON structure.

- **EFI_REST_JSON_RESOURCE_TYPE_IDENTIFIER** set to NULL means the returned structure format may not be in the same form as the consumer’s expectation. The consumer then has to check the **EFI_REST_JSON_RESOURCE_TYPE_IDENTIFIER** for the Redfish namespace and datatype, and use the correct prototype for structure reference.
32 - Secure Boot and Driver Signing

32.1 Secure Boot

This protocol is intended to provide access for generic authentication information associated with
specific device paths. The authentication information is configurable using the defined interfaces.
Successive configuration of the authentication information will overwrite the previously configured
information. Once overwritten, the previous authentication information will not be retrievable.

EFI_AUTHENTICATION_INFO_PROTOCOL

Summary
This protocol is used on any device handle to obtain authentication information associated with the
physical or logical device.

GUID
#define EFI_AUTHENTICATION_INFO_PROTOCOL_GUID \
{0x7671d9d0,0x53db,0x4173,\ 
 {0xaa,0x69,0x23,0x27,0xf2,0x1f,0x0b,0xc7}}

Protocol Interface Structure

typedef struct _EFI_AUTHENTICATION_INFO_PROTOCOL {
EFI_AUTHENTICATION_INFO_PROTOCOL_GET Get;
EFI_AUTHENTICATION_INFO_PROTOCOL_SET Set;
} EFI_AUTHENTICATION_INFO_PROTOCOL;

Parameters
Get() Used to retrieve the Authentication Information associated with the controller handle
Set() Used to set the Authentication information associated with the controller handle

Description
The EFI_AUTHENTICATION_INFO_PROTOCOL provides the ability to get and set the authentication
information associated with the controller handle.

EFI_AUTHENTICATION_INFO_PROTOCOL.Get()

Summary
Retrieves the Authentication information associated with a particular controller handle.
Prototype

typedef
  EFI_STATUS
  (EFIAPI *EFI_AUTHENTICATION_INFO_PROTOCOL_GET) (
    IN EFI_AUTHENTICATION_INFO_PROTOCOL *This,
    IN EFI_HANDLE ControllerHandle,
    OUT VOID **Buffer
  );

Parameters

  This Pointer to the EFI_AUTHENTICATION_INFO_PROTOCOL
  ControllerHandle Handle to the Controller
  Buffer Pointer to the authentication information. This function is responsible for allocating the buffer and it is the caller's responsibility to free buffer when the caller is finished with buffer.

Description

This function retrieves the Authentication Node for a given controller handle.

Status Codes Returned

<table>
<thead>
<tr>
<th>Status Code</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>EFI_SUCCESS</td>
<td>Successfully retrieved Authentication information for the given ControllerHandle</td>
</tr>
<tr>
<td>EFI_INVALID_PARAMETER</td>
<td>No matching Authentication information found for the given ControllerHandle</td>
</tr>
<tr>
<td>EFI_DEVICE_ERROR</td>
<td>The authentication information could not be retrieved due to a hardware error.</td>
</tr>
</tbody>
</table>

EFI_AUTHENTICATION_INFO_PROTOCOL.Set()

Summary

Set the Authentication information for a given controller handle.

Prototype

typedef
  EFI_STATUS
  (EFIAPI *EFI_AUTHENTICATION_INFO_PROTOCOL_SET) (
    IN EFI_AUTHENTICATION_INFO_PROTOCOL *This,
    IN EFI_HANDLE ControllerHandle
    IN VOID *Buffer
  );

Parameters

  This Pointer to the EFI_AUTHENTICATION_INFO_PROTOCOL
  ControllerHandle Handle to the controller.
Buffer  Pointer to the authentication information.

Description
This function sets the authentication information for a given controller handle. If the authentication node exists corresponding to the given controller handle this function overwrites the previously present authentication information.

Status Codes Returned

<table>
<thead>
<tr>
<th>Status Code</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>EFI_SUCCESS</td>
<td>Successfully set the Authentication node information for the given ControllerHandle.</td>
</tr>
<tr>
<td>EFI_UNSUPPORTED</td>
<td>If the platform policies do not allow setting of the Authentication information.</td>
</tr>
<tr>
<td>EFI_DEVICE_ERROR</td>
<td>The authentication node information could not be configured due to a hardware error.</td>
</tr>
<tr>
<td>EFI_OUT_OF_RESOURCES</td>
<td>Not enough storage is available to hold the data.</td>
</tr>
</tbody>
</table>

Authentication Nodes
The authentication node is associated with specific controller paths. There can be various types of authentication nodes, each describing a particular authentication method and associated properties.

Generic Authentication Node Structures
An authentication node is a variable length binary structure that is made up of variable length authentication information. **Table 32-1** defines the generic structure. The Authentication type GUID defines the corresponding authentication node.

Table 32-1 Generic Authentication Node Structure

<table>
<thead>
<tr>
<th>Mnemonic</th>
<th>Byte Offset</th>
<th>Byte Length</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Type GUID</td>
<td>0</td>
<td>16</td>
<td>Authentication Type GUID</td>
</tr>
<tr>
<td>Length</td>
<td>16</td>
<td>2</td>
<td>Length of this structure in bytes.</td>
</tr>
<tr>
<td>Specific Authentication Data</td>
<td>18</td>
<td>n</td>
<td>Specific Authentication Data. Type defines the authentication method and associated type of data. Size of the data is included in the length.</td>
</tr>
</tbody>
</table>

All Authentication Nodes are byte-packed data structures that may appear on any byte boundary. All code references to Authentication Nodes must assume all fields are UNALIGNED. Since every Authentication Node contains a length field in a known place, it is possible to traverse Authentication Node of unknown type.

CHAP (using RADIUS) Authentication Node
This Authentication Node type defines the CHAP authentication using RADIUS information.
GUID

```c
#define EFI_AUTHENTICATION_CHAP_RADIUS_GUID \  
{0xd6062b50,0x15ca,0x11da,\  
{0x92,0x19,0x00,0x10,0x83,0xff,0xca,0x4d}}
```

Node Definition

Table 32-2 CHAP Authentication Node Structure using RADIUS

<table>
<thead>
<tr>
<th>Mnemonic</th>
<th>Byte Offset</th>
<th>Byte Length</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Type</td>
<td>0</td>
<td>16</td>
<td>EFI_AUTHENTICATION_CHAP_RADIUS_GUID</td>
</tr>
<tr>
<td>Length</td>
<td>16</td>
<td>2</td>
<td>Length of this structure in bytes. Total length is $58+P+Q+R+S+T$</td>
</tr>
<tr>
<td>RADIUS IP Address</td>
<td>18</td>
<td>16</td>
<td>Radius IPv4 or IPv6 Address</td>
</tr>
<tr>
<td>Reserved</td>
<td>34</td>
<td>2</td>
<td>Reserved</td>
</tr>
<tr>
<td>NAS IP Address</td>
<td>36</td>
<td>16</td>
<td>NAS IPv4 or IPv6 Address</td>
</tr>
<tr>
<td>NAS Secret Length</td>
<td>52</td>
<td>2</td>
<td>NAS Secret Length P</td>
</tr>
<tr>
<td>NAS Secret</td>
<td>54</td>
<td>P</td>
<td>NAS Secret</td>
</tr>
<tr>
<td>CHAP Secret Length</td>
<td>54+P</td>
<td>2</td>
<td>CHAP Secret Length Q</td>
</tr>
<tr>
<td>CHAP Secret</td>
<td>56+P</td>
<td>q</td>
<td>CHAP Secret</td>
</tr>
<tr>
<td>CHAP Name Length</td>
<td>56 +Q</td>
<td>2</td>
<td>CHAP Name Length R</td>
</tr>
<tr>
<td>CHAP Name</td>
<td>58 +P+Q</td>
<td>r</td>
<td>CHAP Name String</td>
</tr>
<tr>
<td>Reverse CHAP Name Length</td>
<td>58+P+Q+R</td>
<td>2</td>
<td>Reverse CHAP Name length</td>
</tr>
<tr>
<td>Reverse CHAP Name</td>
<td>60+P+Q+R</td>
<td>S</td>
<td>Reverse CHAP Name</td>
</tr>
<tr>
<td>Reverse CHAP Secret Length</td>
<td>60+P+Q+R+S</td>
<td>2</td>
<td>Reverse CHAP Length</td>
</tr>
<tr>
<td>Reverse CHAP Secret</td>
<td>62+P+Q+R+T</td>
<td>T</td>
<td>Reverse CHAP Secret</td>
</tr>
</tbody>
</table>

Summary

- RADIUS IP Address: RADIUS Server IPv4 or IPv6 Address
- NAS IP Address: Network Access Server IPv4 or IPv6 Address (OPTIONAL)
- NAS Secret Length: Network Access Server Secret Length in bytes (OPTIONAL)
- NAS Secret: Network Access Server secret (OPTIONAL)
- CHAP Secret Length: CHAP Initiator Secret length in bytes
- CHAP Secret: CHAP Initiator Secret
- CHAP Name Length: CHAP Initiator Name Length in bytes
- CHAP Name: CHAP Initiator Name
- Reverse CHAP name length: Reverse CHAP name length
- Reverse CHAP Name: Reverse CHAP name
- Reverse CHAP Secret Length: Reverse CHAP secret length
Reverse CHAP Secret

Reverse CHAP secret

CHAP (using local database)

Authentication Node

This Authentication Node type defines CHAP using local database information.

GUID

```c
#define EFI_AUTHENTICATION_CHAP_LOCAL_GUID
{0xc280c73e,0x15ca,0x11da,
{0xb0,0xca,0x00,0x10,0x83,0xff,0xca,0x4d}}
```

Node Definition

Table 32-3 CHAP Authentication Node Structure using Local Database

<table>
<thead>
<tr>
<th>Mnemonic</th>
<th>Byte Offset</th>
<th>Byte Length</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Type</td>
<td>0</td>
<td>16</td>
<td>EFI_AUTHENTICATION_CHAP_LOCAL_GUID</td>
</tr>
<tr>
<td>Length</td>
<td>16</td>
<td>2</td>
<td>Length of this structure in bytes. Total length is 58+P+Q+R+S+T</td>
</tr>
<tr>
<td>Reserved</td>
<td>18</td>
<td>2</td>
<td>Reserved for future use</td>
</tr>
<tr>
<td>User Secret Length</td>
<td>20</td>
<td>2</td>
<td>User Secret Length</td>
</tr>
<tr>
<td>User Secret</td>
<td>22</td>
<td>p</td>
<td>User Secret</td>
</tr>
<tr>
<td>User Name Length</td>
<td>22+p</td>
<td>2</td>
<td>User Name Length</td>
</tr>
<tr>
<td>User Name</td>
<td>24+p</td>
<td>q</td>
<td>User Name</td>
</tr>
<tr>
<td>CHAP Secret Length</td>
<td>24+p+q</td>
<td>2</td>
<td>CHAP Secret Length</td>
</tr>
<tr>
<td>CHAP Secret</td>
<td>26+p+q</td>
<td>r</td>
<td>CHAP Secret</td>
</tr>
<tr>
<td>CHAP Name Length</td>
<td>26+p+q+r</td>
<td>2</td>
<td>CHAP Name Length</td>
</tr>
<tr>
<td>CHAP Name</td>
<td>28+p+q+r</td>
<td>s</td>
<td>CHAP Name String</td>
</tr>
<tr>
<td>Reverse CHAP Name Length</td>
<td>58+P+Q+R</td>
<td>2</td>
<td>Reverse CHAP Name length</td>
</tr>
<tr>
<td>Reverse CHAP Name</td>
<td>60+P+Q+R</td>
<td>S</td>
<td>Reverse CHAP Name</td>
</tr>
<tr>
<td>Reverse CHAP Secret Length</td>
<td>60+P+Q+R+S</td>
<td>2</td>
<td>Reverse CHAP Secret</td>
</tr>
<tr>
<td>Reverse CHAP Secret</td>
<td>62+P+Q+R+S</td>
<td>T</td>
<td>Reverse CHAP Secret</td>
</tr>
</tbody>
</table>

Summary

- User Secret Length
- User Secret
- User Name Length
- User Name
- CHAP Secret Length
- CHAP Secret
- CHAP Name Length

User Secret Length in bytes
User Secret
User Name Length in bytes
User Name
CHAP Initiator Secret length in bytes
CHAP Initiator Secret
CHAP Initiator Name Length in bytes
32.2 UEFI Driver Signing Overview

This section describes a means of generating a digital signature for a UEFI executable, embedding that digital signature within the UEFI executable and verifying that the digital signature is from an authorized source.

The UEFI specification provides a standard format for executables. These executables may be located on un-secured media (such as a hard drive or unprotected flash device) or may be delivered via a un-secured transport layer (such as a network) or originate from a un-secured port (such as ExpressCard device or USB device). In each of these cases, the system provider may decide to authenticate either the origin of the executable or its integrity (i.e., it has not been tampered with). This section describes a means of doing so.

32.2.1 Digital Signatures

As a rule, digital signatures require two pieces: the data (often referred to as the message) and a public/private key pair. In order to create a digital signature, the message is processed by a hashing algorithm to create a hash value. This hash value is, in turn, encrypted using a signature algorithm and the private key to create the digital signature.
In order to verify a signature, two pieces of data are required: the original message and the public key. First, the hash must be calculated exactly as it was calculated when the signature was created. Then the digital signature is decoded using the public key and the result is compared against the computed hash. If the two are identical, then you can be sure that message data is the one originally signed and it has not been tampered with.
32.2.2 Embedded Signatures

The signatures used for digital signing of UEFI executables are embedded directly within the executable itself. Within the header is an array of directory entries. Each of these entries points to interesting places within the executable image. The fifth data directory entry contains a pointer to a list of certificates along with the length of the certificate areas. Each certificate may contain a digital signature used for validating the driver.

The following diagram illustrates how certificates are embedded in the PE/COFF file:
Within the PE/COFF optional header is a data directory. The 5th entry, if filled, points to a list of certificates. Normally, these certificates are appended to the end of the file.

### 32.2.3 Creating Image Digests from Images

One of the pieces required for creating a digital signature is the image digest. For a detailed description on how to create image digests from PE/COFF images, refer to the "Creating the PE Image Hash" section of the Microsoft Authenticode Format specification (see References).

### 32.2.4 Code Definitions

This section describes data structures used for signing UEFI executables.
WIN_CERTIFICATE

Summary
The WIN_CERTIFICATE structure is part of the PE/COFF specification.

Prototype

typedef struct _WIN_CERTIFICATE {
    UINT32 dwLength;
    UINT16 wRevision;
    UINT16 wCertificateType;
    //UINT8 bCertificate[ANYSIZE_ARRAY];
} WIN_CERTIFICATE;

dwLength The length of the entire certificate, including the length of the header, in bytes.
wRevision The revision level of the WIN_CERTIFICATE structure. The current revision level is 0x0200.
wCertificateType The certificate type. See WIN_CERT_TYPE_xxx for the UEFI certificate types. The UEFI specification reserves the range of certificate type values from 0x0EF0 to 0x0EFF.
bCertificate The actual certificate. The format of the certificate depends on wCertificateType. The format of the UEFI certificates is defined below.

Related Definitions
#define WIN_CERT_TYPE_PKCS_SIGNED_DATA 0x0002
#define WIN_CERT_TYPE_EFI_PKCS1_15 0x0EF0
#define WIN_CERT_TYPE_EFI_GUID 0x0EF1

Description
This structure is the certificate header. There may be zero or more certificates. If

• the wCertificateType field is set to WIN_CERT_TYPE_EFI_PKCS1_15, then the certificate follows the format described in WIN_CERTIFICATE_EFI_PKCS1_15.

• If the wCertificateType field is set to WIN_CERT_TYPE_EFI_GUID, then the certificate follows the format described in WIN_CERTIFICATE_EFI_GUID.

• If the wCertificateType field is set to WIN_CERT_TYPE_PKCS_SIGNED_DATA then the certificate is formatted as described in the Authenticode specification.

These certificates can be validated using the contents of the signature database described in Section 32.4.1. The following table illustrates the relationship between the certificates and the signature types in the database.

Note: In the case of a WIN_CERT_TYPE_PKCS_SIGNED_DATA (or WIN_CERT_TYPE_EFI_GUID where CertType = EFI_CERT_TYPE_PKCS7_GUID) certificate, a match can occur against an entry in the authorized signature database (or the forbidden signature database; see Section 32.6.1) at
any level of the chain of X.509 certificates present in the certificate, and matches can occur against any of the applicable signature types defined in Section 32.4:

Table 32-4 PE/COFF Certificates Types and UEFI Signature Database Certificate Types

<table>
<thead>
<tr>
<th>Image Certificate Type</th>
<th>Verified Using Signature Database Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>WIN_CERT_TYPE_EFI_PKCS1_15 (Signature Size = 256 bytes)</td>
<td>EFI_CERT_RSA2048_GUID (public key)</td>
</tr>
<tr>
<td>WIN_CERT_TYPE_EFI_GUID (CertType = EFI_CERT_TYPE_RSA2048_SHA256_GUID)</td>
<td>EFI_CERT_RSA2048_GUID (public key).</td>
</tr>
<tr>
<td>WIN_CERT_TYPE_EFI_GUID (CertType = EFI_CERT_TYPE_PKCS7_GUID)</td>
<td>EFI_CERT_X509_GUID, EFI_CERT_RSA2048_GUID (when applicable)</td>
</tr>
<tr>
<td>WIN_CERT_TYPE_PKCS_SIGNED_DATA</td>
<td>EFI_CERT_X509_GUID, EFI_CERT_RSA2048_GUID (when applicable)</td>
</tr>
<tr>
<td>(Always applicable regardless of whether a certificate is present or not)</td>
<td>EFI_CERT_SHA1_GUID, EFI_CERT_SHA224_GUID, EFI_CERT_SHA256_GUID, EFI_CERT_SHA384_GUID, EFI_CERT_SHA512_GUID</td>
</tr>
</tbody>
</table>

**WIN_CERTIFICATE_EFI_PKCS1_15**

**Summary**
Certificate which encapsulates the RSASSA_PKCS1-v1_5 digital signature.
Prototype

typedef struct _WIN_CERTIFICATE_EFI_PKCS1_15 {
    WIN_CERTIFICATE Hdr;
    EFI_GUID HashAlgorithm;
    // UINT8 Signature[ANYSIZE_ARRAY];
} WIN_CERTIFICATE_EFI_PKCS1_15;

Hdr This is the standard WIN_CERTIFICATE header, where wCertificateType is set to WIN_CERT_TYPE_EFI_PKCS1_15.

HashAlgorithm This is the hashing algorithm which was performed on the UEFI executable when creating the digital signature. It is one of the enumerated values pre-defined in Section 37.1.2.1. See EFI_HASH_ALGORITHM_x.

Signature This is the actual digital signature. The size of the signature is the same size as the key (2048-bit key is 256 bytes) and can be determined by subtracting the length of the other parts of this header from the total length of the certificate as found in Hdr.dwLength.

Description

The WIN_CERTIFICATE_EFI_PKCS1_15 structure is derived from WIN_CERTIFICATE and encapsulates the information needed to implement the RSASSA-PKCS1-v1_5 digital signature algorithm as specified in RFC2437, sections 8-9.

WIN_CERTIFICATE_EFI_GUID

Summary

Certificate which encapsulates a GUID-specific digital signature.

Prototype

typedef struct _WIN_CERTIFICATE_EFI_GUID {
    WIN_CERTIFICATE Hdr;
    EFI_GUID CertType;
    UINT8 CertData[ANYSIZE_ARRAY];
} WIN_CERTIFICATE_EFI_GUID;

Hdr This is the standard WIN_CERTIFICATE header, where wCertificateType is set to WIN_CERT_TYPE_EFI_GUID.

CertType This is the unique id which determines the format of the CertData.

CertData This is the certificate data. The format of the data is determined by the CertType.
Related Definitions

```
#define EFI_CERT_TYPE_RSA2048_SHA256_GUID
{0xa7717414, 0xc616, 0x4977, \n{0x94, 0x20, 0x84, 0x47, 0x12, 0xa7, 0x35, 0xbf}}
#define EFI_CERT_TYPE_PKCS7_GUID
{0x4aafd29d, 0x68df, 0x49ee, \n{0x8a, 0xa9, 0x34, 0x7d, 0x37, 0x56, 0x65, 0xa7}}
typedef struct _EFI_CERT_BLOCK_RSA_2048_SHA256 {
  EFI_GUID HashType;
  UINT8  PublicKey[256];
  UINT8  Signature[256];
} EFI_CERT_BLOCK_RSA_2048_SHA256;
```

Public Key

The RSA exponent e for this structure is 0x10001.

Signature

This signature block is PKCS 1 version 1.5 formatted.

Description

The WIN_CERTIFICATE_UEFI_GUID certificate type allows new types of certificates to be developed for driver authentication without requiring a new certificate type. The CertType defines the format of the CertData, which length is defined by the size of the certificate less the fixed size of the WIN_CERTIFICATE_UEFI_GUID structure.

- If CertType is EFI_CERT_TYPE_RSA2048_SHA256_GUID then the structure which follows has the format specified by EFI_CERT_BLOCK_RSA_2048_SHA256.
- If CertType is EFI_CERT_TYPE_PKCS7_GUID then the CertData component shall contain a DER-encoded PKCS #7 version 1.5 [RFC2315] SignedData value.

32.3 Firmware/OS Key Exchange: creating trust relationships

This section describes a means of creating a trust relationship between the platform owner, the platform firmware, and an operating system. This trust relationship enables the platform firmware and one or more operating systems to exchange information in a secure manner.

The trust relationship uses two types of asymmetric key pairs:

**Platform Key (PK)**

The platform key establishes a trust relationship between the platform owner and the platform firmware. The platform owner enrolls the public half of the key (PK_{pub}) into the platform firmware. The platform owner can later use the private half of the key (PK_{priv}) to change platform ownership or to enroll a Key Exchange Key. For UEFI, the recommended Platform Key format is RSA-2048. See “Enrolling The Platform Key” and “Clearing The Platform Key” for more information.

**Key Exchange Key (KEK)**

Key exchange keys establishes a trust relationship between the operating system and the platform firmware. Each operating system (and potentially, each 3rd party application which need to communicate with platform firmware) enrolls a public key
(KEKpub) into the platform firmware. See “Enrolling Key Exchange Keys” for more information.

While no Platform Key is enrolled, the SetupMode variable shall be equal to 1. While SetupMode == 1, the platform firmware shall not require authentication in order to modify the Platform Key, Key Enrollment Key, OsRecoveryOrder, OsRecovery####, and image security databases.

After the Platform Key is enrolled, the SetupMode variable shall be equal to 0. While SetupMode == 0, the platform firmware shall require authentication in order to modify the Platform Key, Key Enrollment Key, OsRecoveryOrder, OsRecovery####, and image security databases.

While no Platform Key is enrolled, and while the variable AuditMode == 0, the platform is said to be operating in setup mode.

After the Platform Key is enrolled, and while the variable AuditMode == 0, the platform is operating in user mode. The platform will continue to operate in user mode until the Platform Key is cleared, or the system is transitioned to either Audit or Deployed Modes. See "Clearing The Platform Key," "Transitioning to Audit Mode," and "Transitioning to Deployed Mode" for more information.

Audit Mode enables programmatic discovery of signature list combinations that successfully authenticate installed EFI images without the risk of rendering a system unbootable. Chosen signature lists configurations can be tested to ensure the system will continue to boot after the system is transitioned out of Audit Mode. Details on how to transition to Audit Mode are detailed below in the section "Transitioning to Audit Mode." After transitioning to Audit Mode, signature enforcement is disabled such that all images are initialized and enhanced Image Execution Information Table (IEIT) logging is performed including recursive validation for multi-signed images.

Deployed Mode is the most secure mode. For details on transitioning to Deployed Mode see the section "Transitioning to Deployed Mode" below. By design, both User Mode and Audit Mode support unauthenticated transitions to Deployed Mode. However, to move from Deployed Mode to any other mode requires a secure platform-specific method, or deleting the PK, which is authenticated.

Secure Boot Mode transitions to User Mode or Deployed Mode shall take effect immediately. Mode transitions to Setup Mode or Audit Mode may either take effect immediately (recommended) or after a reset. For implementations that require a reset, the mode transition shall be processed prior to the initialization of the SecureBoot variable, and the SetVariable() workflow shall be as follows:

1. If the variable has an authenticated attribute, it shall be authenticated as specified, and failure will result in immediate termination of this workflow by returning the appropriate error.
2. Check secure storage to determine if a Secure Boot Mode transition is already queued. If a transition is already queued, terminate this workflow by returning EFI_ALREADY_STARTED
3. Queue the request to secure storage
4. The Secure Boot Mode and Policy variables SHALL remain unchanged
5. Return EFI_WARN_RESET_REQUIRED.
6. After reboot, if the transition is successful, Secure Boot Mode and Policy variables will change accordingly. If the transition to lower security modes is rejected or fail, the workflow is terminated and the Secure Boot Mode and Policy variables remain unchanged
32.3.1 Enrolling The Platform Key

The platform owner enrolls the public half of the Platform Key (PK\textsubscript{pub}) by calling the UEFI Boot Service \texttt{SetVariable()} as specified in Section 8.2.1. If the platform is in setup mode, then the new PK\textsubscript{pub} may be signed with its PK\textsubscript{priv} counterpart. If the platform is in user mode, then the new PK\textsubscript{pub} must be signed with the current PK\textsubscript{priv}. When the platform is in setup mode, a successful enrollment of a Platform Key shall cause the platform to immediately transition to user mode.

The authenticated PK variable can always be read but can only be written if the platform is in setup mode, or if the platform is in user mode and the provided PK\textsubscript{pub} is signed with the current PK\textsubscript{priv}.

The name and GUID of the Platform Key variable are specified in Section 3.3 “Globally Defined Variables” The variable has the format of a signature database as described in “Signature Database” below, with exactly one entry.

The platform vendor may provide a default PKpub in the PKDefault variable described in Section 3.3. This variable is formatted identically to the Platform Key variable. If present, this key may optionally be used as the public half of the Platform Key when transitioning from setup mode to user mode. If so, it may be
read, placed within an `EFI_VARIABLE_AUTHENTICATION2` structure and copied to the Platform Key variable using the `SetVariable()` call.

### 32.3.2 Clearing The Platform Key

The platform owner clears the public half of the Platform Key (PK\textsubscript{pub}) by deleting the Platform Key variable using UEFI Runtime Service `SetVariable()`. The data buffer submitted to the `SetVariable()` must be signed with the current PK\textsubscript{priv}; see Section 8.1.1 for details. The name and GUID of the Platform Key variable are specified in Section 3.3, “Globally Defined Variables”.

The platform key may also be cleared using a secure platform-specific method. When platform key is cleared, the global variable SetupMode must also be updated to 1.

### 32.3.3 Transitioning to Audit Mode

To enter Audit Mode, a new UEFI variable AuditMode is set to 1. Entering Audit Mode has the side effect of changing SetupMode == 1, PK is cleared, and the new DeployedMode == 0.

**Note:** The AuditMode variable is only writable before `ExitBootServices()` is called when the system is not in Deployed Mode. See Figure 32-4 for more details.

### 32.3.4 Transitioning to Deployed Mode

To enter Deployed Mode from Audit Mode, set the variable PK. To enter Deployed Mode from User Mode, set the variable DeployedMode to 1. This transition takes effect immediately with no reset required. Entering Deployed Mode has the side effect of changing SetupMode == 0, AuditMode == 0 and is made read-only, and DeployedMode == 1 and is made read-only. See Figure 32-4 for more details.

### 32.3.5 Enrolling Key Exchange Keys

Key exchange keys are stored in a signature database as described in "Signature Database" below. The signature database is stored as an authenticated UEFI variable.

The platform owner enrolls the key exchange keys by either calling `SetVariable()` as specified in Section 8.2.1 with the `EFI_VARIABLE_APPEND_WRITE` attribute set and the Data parameter containing the new key(s), or by reading the database using `GetVariable()`, appending the new key exchange key to the existing keys and then writing the database using `SetVariable()` as specified in Section 8.2.1 without the `EFI_VARIABLE_APPEND_WRITE` attribute set.

The authenticated UEFI variable that stores the key exchange keys (KEKs) can always be read but only be written if:

- The platform is in user mode and the provided variable data is signed with the current PK\textsubscript{priv}, or
- The platform is in setup mode (in this case the variable can be written without a signature validation, but the `SetVariable()` call needs to be formatted in accordance with the procedure for authenticated variables in Section 8.2.1).

The name and GUID of the Key Exchange Key variable are specified in Section 3.3, “Globally Defined Variables.”
The platform vendor may provide a default set of Key Exchange Keys in the KEKDefault variable described in Section 3.3. If present, these keys (or a subset) may optionally be used when performing the initial enrollment of Key Exchange Keys. If any are to be used, they may be parsed from the variable and enrolled as described above.

32.3.6 Platform Firmware Key Storage Requirements

This section describes the platform firmware storage requirements of the different types of keys.

**Platform Keys:**

The public key must be stored in non-volatile storage which is tamper and delete resistant.

**Key Exchange Keys:**

The public key must be stored in non-volatile storage which is tamper resistant.

Careful consideration should be given to the security and configuration of any out-of-band management agent (e.g. hypervisor or service processor) such that the platform cannot exploit the management agent in order to circumvent Secure Boot.

32.4 Firmware/OS Key Exchange: passing public keys

This section describes a means of passing public keys from the OS to the platform firmware so that these keys can be used to securely pass information between the OS and the platform firmware.

Typically, the OS has been unable to communicate sensitive information or enforce any sort of policy because of the possibility of spoofing by a malicious software agent. That is, the platform firmware has been unable to trust the OS. By enrolling these public keys, authorized by the platform owner, the platform firmware can now check the signature of data passed by the operating system.

Of course if the malicious software agent is running as part of the OS, such as a rootkit, then any communication between the firmware and operating system still remains the subject of spoofing as the malicious code has access to the key exchange key.

32.4.1 Signature Database

**EFI_SIGNATURE_DATA**

**Summary**

The format of a signature database.

**Prototype**

```c
#pragma pack(1)
typedef struct _EFI_SIGNATURE_DATA {
    EFI_GUID    SignatureOwner;
    UINT8       SignatureData[...];
} EFI_SIGNATURE_DATA;
```
typedef struct _EFI_SIGNATURE_LIST {
    EFI_GUID   SignatureType;
    UINT32    SignatureListSize;
    UINT32    SignatureHeaderSize;
    UINT32    SignatureSize;
    // UINT8 SignatureHeader[SignatureHeaderSize];
    // EFI_SIGNATURE_DATA Signatures[...][SignatureSize];
} EFI_SIGNATURE_LIST;

#pragma pack()

Members

SignatureListSize
Total size of the signature list, including this header.

SignatureType
Type of the signature. GUID signature types are defined in "Related Definitions" below.

SignatureHeaderSize
Size of the signature header which precedes the array of signatures.

SignatureSize
Size of each signature. Must be at least the size of EFI_SIGNATURE_DATA.

SignatureHeader
Header before the array of signatures. The format of this header is specified by the SignatureType.

Signatures
An array of signatures. Each signature is SignatureSize bytes in length. The format of the signature is defined by the SignatureType.

SignatureOwner
An identifier which identifies the agent which added the signature to the list.

Description
The signature database consists of zero or more signature lists. The size of the signature database can be determined by examining the size of the UEFI variable.

Each signature list is a list of signatures of one type, identified by SignatureType. The signature list contains a header and then an array of zero or more signatures in the format specified by the header. The size of each signature in the signature list is specified by SignatureSize.

Each signature has an owner SignatureOwner, which is a GUID identifying the agent which inserted the signature in the database. Agents might include the operating system or an OEM-supplied driver or application. Agents may examine this field to understand whether they should manage the signature or not.
Figure 32-5 Signature lists

### Related Definitions

```c
#define EFI_CERT_SHA256_GUID \
  { 0xc1c41626, 0x504c, 0x4092, \
    { 0xac, 0xa9, 0x41, 0xf9, 0x36, 0x93, 0x43, 0x28 } };  
```

This identifies a signature containing a SHA-256 hash. The `SignatureHeader` size shall always be 0. The `SignatureSize` shall always be 16 (size of `SignatureOwner` component) + 32 bytes.

```c
#define EFI_CERT_RSA2048_GUID \
  { 0x3c5766e8, 0x269c, 0x4e34, \
    { 0xaa, 0x14, 0xed, 0x77, 0x6e, 0x85, 0xb3, 0xb6 } };  
```

This identifies a signature containing an RSA-2048 key. The key (only the modulus since the public key exponent is known to be 0x10001) shall be stored in big-endian order.

The `SignatureHeader` size shall always be 0. The `SignatureSize` shall always be 16 (size of `SignatureOwner` component) + 256 bytes.

```c
#define EFI_CERT_RSA2048_SHA256_GUID \
  { 0xe2b36190, 0x879b, 0x4a3d, \
    { 0xad, 0x8d, 0xf2, 0xe7, 0xbb, 0xa3, 0x27, 0x84 } };  
```

This identifies a signature containing a RSA-2048 signature of a SHA-256 hash. The `SignatureHeader` size shall always be 0. The `SignatureSize` shall always be 16 (size of `SignatureOwner` component) + 256 bytes.
#define EFI_CERT_SHA1_GUID \
{ 0x826ca512, 0xcf10, 0x4ac9, \
{ 0xb1, 0x87, 0xbe, 0x01, 0x49, 0x66, 0x31, 0xbd } };

This identifies a signature containing a SHA-1 hash. The `SignatureSize` shall always be 16 (size of `SignatureOwner` component) + 20 bytes.

#define EFI_CERT_RSA2048_SHA1_GUID \
{ 0x67f8444f, 0x8743, 0x48f1, \
{ 0xa3, 0x28, 0x1e, 0xaa, 0xb8, 0x73, 0x60, 0x80 } };

This identifies a signature containing a RSA-2048 signature of a SHA-1 hash. The `SignatureHeader` size shall always be 0. The `SignatureSize` shall always be 16 (size of `SignatureOwner` component) + 256 bytes.

#define EFI_CERT_X509_GUID \
{ 0xa5c059a1, 0x94e4, 0x4aa7, \
{ 0x87, 0xb5, 0xab, 0x15, 0x5c, 0x2b, 0xf0, 0x72 } };

This identifies a signature based on a DER-encoded X.509 certificate. If the signature is an X.509 certificate then verification of the signature of an image should validate the public key certificate in the image using certificate path verification, up to this X.509 certificate as a trusted root. The `SignatureHeader` size shall always be 0. The `SignatureSize` may vary but shall always be 16 (size of the `SignatureOwner` component) + the size of the certificate itself.

Note: This means that each certificate will normally be in a separate `EFI_SIGNATURE_LIST`.

#define EFI_CERT_SHA224_GUID \
{ 0xb6e5233, 0xa65c, 0x44c9, \
{0x94, 0x07, 0xd9, 0xab, 0x83, 0xbf, 0xc8, 0xbd} };

This identifies a signature containing a SHA-224 hash. The `SignatureHeader` size shall always be 0. The `SignatureSize` shall always be 16 (size of `SignatureOwner` component) + 28 bytes.

#define EFI_CERT_SHA384_GUID \
{ 0xff3e5307, 0x9fd0, 0x48c9, \
{0x85, 0xf1, 0x8a, 0xd5, 0x6c, 0x70, 0x1e, 0x01})};

This identifies a signature containing a SHA-384 hash. The `SignatureHeader` size shall always be 0. The `SignatureSize` shall always be 16 (size of `SignatureOwner` component) + 48 bytes.
#define EFI_CERT_SHA512_GUID \
{ 0x93e0fae, 0xa6c4, 0x4f50, \
{0x9f, 0x1b, 0xd4, 0x1e, 0x2b, 0x89, 0xc1, 0x9a}}

This identifies a signature containing a SHA-512 hash. The SignatureHeader size shall always be 0. The SignatureSize shall always be 16 (size of SignatureOwner component) + 64 bytes.

#define EFI_CERT_X509_SHA256_GUID \
{ 0x3bd2a492, 0x96c0, 0x4079, \
{ 0xb4, 0x20, 0xfc, 0xf9, 0xe, 0xf1, 0x03, 0xed } };

Prototype
#pragma pack(1)
typedef struct _EFI_CERT_X509_SHA256 {
  EFI_SHA256_HASH ToBeSignedHash;
  EFI_TIME TimeOfRevocation;
} EFI_CERT_X509_SHA256;
#pragma pack()

Members
ToBeSignedHash
The SHA256 hash of an X.509 certificate’s To-Be-Signed contents.

TimeOfRevocation
The time that the certificate shall be considered to be revoked.

This identifies a signature containing the SHA256 hash of an X.509 certificate’s To-Be-Signed contents, and a time of revocation. The SignatureHeader size shall always be 0. The SignatureSize shall always be 16 (size of the SignatureOwner component) + 48 bytes for an EFI_CERT_X509_SHA256 structure. If the TimeOfRevocation is non-zero, the certificate should be considered to be revoked from that time and onwards, and otherwise the certificate shall be considered to always be revoked.
#define EFI_CERT_X509_SHA384_GUID \
{ 0x7076876e, 0x80c2, 0x4ee6, \
{ 0xaa, 0xd2, 0x28, 0xb3, 0x49, 0xa6, 0x86, 0x5b } }; 

Prototype

```c
#pragma pack(1)
typedef struct _EFI_CERT_X509_SHA384 {
    EFI_SHA384_HASH ToBeSignedHash;
    EFI_TIME      TimeOfRevocation;
} EFI_CERT_X509_SHA384;
#pragma pack()
```

Members

- **ToBeSignedHash**
  - The SHA384 hash of an X.509 certificate’s To-Be-Signed contents.
- **TimeOfRevocation**
  - The time that the certificate shall be considered to be revoked.

This identifies a signature containing the SHA384 hash of an X.509 certificate’s To-Be-Signed contents, and a time of revocation. The `SignatureHeader` size shall always be 0. The `SignatureSize` shall always be 16 (size of the `SignatureOwner` component) + 64 bytes for an `EFI_CERT_X509_SHA384` structure. If the `TimeOfRevocation` is non-zero, the certificate should be considered to be revoked from that time and onwards, and otherwise the certificate shall be considered to always be revoked.

#define EFI_CERT_X509_SHA512_GUID \
{ 0x446dbf63, 0x2502, 0x4cda, \
{ 0xbc, 0xfa, 0x24, 0x65, 0xd2, 0xb0, 0xfe, 0x9d } }; 

Prototype

```c
#pragma pack(1)
typedef struct _EFI_CERT_X509_SHA512 {
    EFI_SHA512_HASH ToBeSignedHash;
    EFI_TIME      TimeOfRevocation;
} EFI_CERT_X509_SHA512;
#pragma pack()
```

Members

- **ToBeSignedHash**
  - The SHA512 hash of an X.509 certificate’s To-Be-Signed contents.
- **TimeOfRevocation**
  - The time that the certificate shall be considered to be revoked.

This identifies a signature containing the SHA512 hash of an X.509 certificate’s To-Be-Signed contents, and a time of revocation. The `SignatureHeader` size shall always be 0. The `SignatureSize` shall always be 16 (size of the `SignatureOwner` component) + 80 bytes for an `EFI_CERT_X509_SHA512`
structure. If the TimeOfRevocation is non-zero, the certificate should be considered to be revoked from that time and onwards, and otherwise the certificate shall be considered to always be revoked.

```c
#define EFI_CERT_EXTERNAL_MANAGEMENT_GUID \
{ 0x452e8ced, 0xdfff, 0x4b8c, \
  { 0xae, 0x01, 0x51, 0x18, 0x86, 0x2e, 0x68, 0x2c } };```

This SignatureType describes a pseudo-signature which will not facilitate authentication. It is only meaningful within a signature list used for authenticating writes through SetVariable(), and is only effective if it is the only signature present in that signature list. It allows a signature list to be populated without providing any means for SetVariable() to succeed. This signature type is intended for use on a platform with an external out-of-band management agent (e.g. hypervisor or service processor). When a platform is configured such that only signatures of this SignatureType are available for authenticating writes to a variable, that variable may only be modified by the external management agent using a platform-specific interface.

When a write may be authenticated using any signature from multiple signature lists, the presence of this signature in one of those signature lists does not inhibit the use of signatures present in the other signature lists. For example, if this signature is placed in PK, an attempt to write to db using SetVariable() will still succeed if it is signed by a valid KEKpriv, but a write to PK or KEK through SetVariable() cannot succeed because no PKpriv exists.

The SignatureHeader size shall always be 0. The SignatureSize shall always be 16 (size of SignatureOwner component) + 1 byte. The one byte of SignatureData exists only for compatibility reasons; It should be written as zero, and any value read should be ignored.

### 32.4.2 Image Execution Information Table

**Summary**

When AuditMode==0, if the image’s signature is not found in the authorized database, or is found in the forbidden database, the image will not be started and instead, information about it will be placed in the **EFI_IMAGE_EXECUTION_INFO_TABLE** (see Section 32.5.3.1, Using The Image Execution Information Table).

When AuditMode==1, an **EFI_IMAGE_EXECUTION_INFO** element is created in the **EFI_IMAGE_EXECUTION_INFO_TABLE** for every certificate found in the certificate table of every image that is validated.

Additionally for every image, an element will be created in the table for every EFI_CERT_SHAXXX that is supported by the platform. The contents of Action for each element are determined by comparing that specific element’s Signature (which will contain exactly 1 **EFI_SIGNATURE_DATA**) to the currently-configured image security databases and policies, and shall be either **EFI_IMAGE_EXECUTION_AUTH_SIG_PASSED**, **EFI_IMAGE_EXECUTION_AUTH_SIG_FAILED**, or **EFI_IMAGE_EXECUTION_POLICY_FAILED**.

Finally, because the system is in Audit Mode, all modules are initialized even if they fail to authenticate, and the **EFI_IMAGE_EXECUTION_INITIALIZED** bit shall be set in Action for all elements.
Prototype

typedef struct {
    EFI_IMAGE_EXECUTION_ACTION Action;
    UINT32 InfoSize;
    // CHAR16 Name[...];
    // EFI_DEVICE_PATH_PROTOCOL DevicePath;
    // EFI_SIGNATURE_LIST Signature;
} EFI_IMAGE_EXECUTION_INFO;

Parameters

Action

Describes the action taken by the firmware regarding this image. Type
EFI_IMAGE_EXECUTION_ACTION is described in “Related Definitions” below.

InfoSize

Size of all of the entire structure.

Name

If this image was a UEFI device driver (for option ROM, for example) this is the null-terminated, user-friendly name for the device. If the image was for an application, then this is the name of the application. If this cannot be determined, then a simple NULL character should be put in this position.

DevicePath

Image device path. The image device path typically comes from the Loaded Image Device Path Protocol installed on the image handle. If image device path cannot be determined, a simple end-of-path device node should be put in this position.

Signature

Zero or more image signatures. If the image contained no signatures, then this field is empty. The type WIN_CERTIFICATE is defined in chapter 26.

Prototype

typedef struct {
    UINTN NumberOfImages;
    EFI_IMAGE_EXECUTION_INFO InformationInfo[...]
} EFI_IMAGE_EXECUTION_INFO_TABLE;

NumberOfImages Number of EFI_IMAGE_EXECUTION_INFO structures.
InformationInfo NumberOfImages instances of EFI_IMAGE_EXECUTION_INFO structures.
Related Definitions

typedef UINT32 EFI_IMAGE_EXECUTION_ACTION;

#define EFI_IMAGE_EXECUTION_AUTHENTICATION 0x00000007
#define EFI_IMAGE_EXECUTION_AUTH_UNTESTED 0x00000000
#define EFI_IMAGE_EXECUTION_AUTH_SIG_FAILED 0x00000001
#define EFI_IMAGE_EXECUTION_AUTH_SIG_PASSED 0x00000002
#define EFI_IMAGE_EXECUTION_AUTH_SIG_NOT_FOUND 0x00000003
#define EFI_IMAGE_EXECUTION_AUTH_SIG_FOUND 0x00000004
#define EFI_IMAGE_EXECUTION_POLICY_FAILED 0x00000005

#define EFI_IMAGE_EXECUTION_INITIALIZED 0x00000008

Description

This structure describes an image in the EFI System Configuration Table. It is only required in the case where image signatures are being checked and the image was not initialized because its signature failed, when AuditMode==1, or was not found in the signature database and an authorized user or the owner would not authorize its execution. It may be used in other cases as well.

In these cases, the information about the image is copied into the EFI System Configuration Table. Information about other images which were successfully initialized may also be included as well, but this is not required.

The Action field describes what action the firmware took with regard to the image and what other information it has about the image, including the device which it is related to.

First, this field describes the results of the firmware’s attempt to authenticate the image.

1. If EFI_IMAGE_EXECUTION_AUTH_UNTESTED is set, then no authentication attempt was made.
2. If EFI_IMAGE_EXECUTION_AUTH_SIG_FAILED is set, then the image had at least one digital signature and the check of the digital signatures failed.
3. If EFI_IMAGE_EXECUTION_AUTH_SIG_PASSED is set, then the image had at least one valid digital signature and a check of that digital signature passed.
4. If EFI_IMAGE_EXECUTION_AUTH_SIG_NOT_FOUND is set, then the image’s signature could not be found in the signature database.
5. If EFI_IMAGE_EXECUTION_AUTH_SIG_FOUND is set, then the image’s signature was found in the signature database.
6. If EFI_IMAGE_EXECUTION_POLICY_FAILED is set, then authentication failed because of (unspecified) firmware security policy.

Second, this field describes whether the image was initialized or not.

This table can be used by an agent which executes later to audit which images were not loaded and perhaps query other sources to discover whether the image should be authorized. If so, the agent can use the method described in “Signature Database Update” to update the Signature Database with the image’s signature. Switching the system into Audit Mode generates a more verbose table which provides additional insights to this agent.
If an attempt to boot a legacy non-UEFI OS takes place when the system is in User Mode, the OS load shall fail and a corresponding `EFI_IMAGE_EXECUTION_INFO` entry shall be created with Action set to `EFI_IMAGE_EXECUTION_AUTH_UNTESTED`, Name set to the NULL-terminated “Description String” from the BIOS Boot Specification Device Path and DevicePath set to the BIOS Boot Specification Device Path (see Section 10.3.6).

### 32.5 UEFI Image Validation

#### 32.5.1 Overview

This section describes a way to use the platform ownership model described in the previous section and the key exchange mechanism to allow the firmware to authenticate a UEFI image, such as an OS loader or an option ROM, using the digital signing mechanisms described here.

The hand-off between the platform firmware and the operating system is a critical part of ensuring secure boot. Since there are large numbers of operating systems and a large number of minor variations in the loaders for those operating systems, it is difficult to carry all possible keys or signatures within the firmware as it ships. This requires some sort of update mechanism, to identify the proper loader. But, as with any update mechanism, there is the risk of allowing malicious software to “authenticate” itself, posing as the real operating system.

Likewise, there are a large number of potential 3rd-party UEFI applications, drivers and option ROMs and it is difficult to carry all possible keys or signatures within the firmware as it ships.

The mechanism described here requires that the platform firmware maintain a signature database, with entries for each authorized UEFI image (the authorized UEFI signature database). The signature database is a single UEFI Variable.

It also requires that the platform firmware maintain a signature database with entries for each forbidden UEFI image. This signature database is also a single UEFI variable.

The signature database is checked when the UEFI Boot Manager is about to start a UEFI image. If the UEFI image’s signature is not found in the authorized database, or is found in the forbidden database, the UEFI image will be deferred and information placed in the Image Execution Information Table. In the case of OS Loaders, the next boot option will be selected. The signature databases may be updated by the firmware, by a pre-OS application or by an OS application or driver.

If a firmware supports the `EFI_CERT_X509_SHA*_GUID` signature types, it should support the RFC3161 timestamp specification. Images whose signature matches one of these types in the forbidden signature database shall only be considered forbidden if the firmware either does not support timestamp verification, or the signature type has a time of revocation equal to zero, or the timestamp does not pass verification against the authorized timestamp and forbidden signature databases, or finally the signature type's time of revocation is less than or equal to the time recorded in the image signature's timestamp. If the timestamp's signature is authorized by the authorized timestamp database and the time recorded in the timestamp is less than the time of revocation, the image shall not be considered forbidden provided it is not forbidden by any other entry in the forbidden signature database. Finally, this requires that firmware supporting timestamp verification must support the authorized timestamp database and have a suitable time stamping authority certificate in that database.
32.5.2 Authorized User

An authorized user (for the purposes of UEFI image security) is one who possesses a key exchange key (KEK_{priv}). This key is used to sign updates to the signature databases.

32.5.3 Signature Database Update

The Authorized, Forbidden, Timestamp, and Recovery signature databases are stored as UEFI authenticated variables (see Variable Services in Section 8.1.1) with the GUID $\text{EFI\_IMAGE\_SECURITY\_DATABASE\_GUID}$ and the names $\text{EFI\_IMAGE\_SECURITY\_DATABASE}$, $\text{EFI\_IMAGE\_SECURITY\_DATABASE1}$, $\text{EFI\_IMAGE\_SECURITY\_DATABASE2}$, and $\text{EFI\_IMAGE\_SECURITY\_DATABASE3}$, respectively.

These authenticated UEFI variables that store the signature databases (db, dbx, dbt, or dbt) can always be read but can only be written if:

- The platform is in user mode and the provided variable data is signed with the private half of a previously enrolled key exchange key (KEK_{priv}), or the platform private key (PK_{priv});

or if

- The platform is in setup mode (in this case the variables can be written without a signature validation, but the SetVariable() call needs to be formatted in accordance with the procedure for authenticated variables in Section 8.2.1)

The signature databases are in the form of Signature Databases, as described in “Signature Database” above.

The platform vendor may provide a default set of entries for the Signature Database in the dbDefault, dbxDefault, dbtDefault, and dbtDefault variables described in Section 3.3. If present, these keys (or a subset) may optionally be used when performing the initial enrollment of signature database entries. If any are to be used, they may be parsed from the variable and enrolled as described below.

If, when adding a signature to the signature database, SetVariable() returns EFI\_OUT\_OF\_RESOURCES, indicating there is no more room, the updater may discard the new signature or it may decide to discard one of the database entries. These authenticated UEFI variables that store the signature databases (db, or dbx, dbt, or dbt) can always be read but can only be written if:
The following diagram illustrates the process for adding a new signature by the OS or an application that has access to a previously enrolled key exchange key using `SetVariable()`. In the diagram, the `EFI_VARIABLE_APPEND_WRITE` attribute is not used. If `EFI_VARIABLE_APPEND_WRITE` had been used, then steps 2 and 3 could have been omitted and step 7 would have included setting the `EFI_VARIABLE_APPEND_WRITE` attribute.

1. The procedure begins by generating a new signature, in the format described by the Signature Database.
2. Call `GetVariable()` using `EFI_IMAGE_SECURITY_DATABASE_GUID` for the `VendorGuid` parameter and `EFI_IMAGE_SECURITY_DATABASE` for the `VariableName` parameter.
3. If the variable exists, go to step 5.
4. Create an empty authorized signature database.
5. Create a new buffer which contains the authorized signature database, along with the new signature appended to the end.
6. Sign the new signature database using the private half of the Key Exchange Key as described in `SetVariable()`.
7. Update the authorized signature database using the UEFI Runtime Service `SetVariable()`.
8. If there was no error, go to step 11.
9. If there was an error because of no more resources, determine whether the database can be shrunk any more. The algorithm by which an agent decides which signatures may be safely removed is agent-specific. In most cases, agents should not remove signatures where the `SignatureOwner` field is not the agent’s. If not, then go to step 11, discarding the new signature.
10. If the signature database could be shrunk further, then remove the entries and go to step 6.
11. Exit.
Figure 32-6 Process for adding a new signature by the OS
32.5.3.1 Using The Image Execution Information Table

During the process of loading UEFI images, the firmware must gather information about which UEFI images were not started. The firmware may additionally gather information about UEFI images which were started. The information is used to create the EFI_IMAGE_EXECUTION_INFO_TABLE, which is added to the EFI System Configuration Table and assigned the GUID EFI_IMAGE_SECURITY_DATABASE_GUID.

For each UEFI image, the following information is collected:

- The image hash.
- The user-friendly name of the UEFI image (if known)
- The device path
- The action taken on the device (was it initialized or why was it rejected)

For more information, see the Image Execution Information Table definition in Section 32.4.2.

32.5.3.2 Firmware Policy

The firmware may approve UEFI images for other reasons than those specified here. For example: whether the image is in the system flash, whether the device providing the UEFI image is secured (in a case, etc.) or whether the image contains another type of platform-supported digital signature.

32.5.3.3 Authorization Process

This section describes the process by which an unknown UEFI image might be authorized to run. Implementations are not required to support all portions of this. For example, an implementation might defer all UEFI image or none.
Table 32-5 Authorization process flow

1. Reset

2. KEY STORE INITIALIZED

3. UEFI APPLICATION VALIDATED?
   - Yes: Go to NEXT APPLICATION
   - No: 7. GO TO NEXT BOOT OPTION

4. START UEFI APPLICATION

5. UEFI APPLICATION APPROVED?
   - Yes: Go to NEXT APPLICATION
   - No: Deferral

6. UEFI APP SIGNATURE ADDED TO DATABASE

7. GO TO NEXT APPLICATION

8. UEFI APPLICATION HASH PASSED IN SYSTEM CONFIGURATION TABLE

9. OS APP VALIDATES UEFI APPLICATION
   - Yes: 10. UEFI APP SIGNATURE DATABASE UPDATED
   - No: End

10. UEFI APP SIGNATURE DATABASE UPDATED

11. End
1. Reset. This is when the platform begins initialization during boot.
2. Key Store Initialization. During the firmware initialization and before any signed UEFI images are initialized, the platform firmware must validate the signature database.
3. UEFI Image Validation Succeeded? During initialization of an UEFI image, the UEFI Boot Manager decides whether or not the UEFI image should be initialized. By comparing the calculated UEFI image signature against that in one of the signature databases, the firmware can determine if there is a match.

The security database $\text{db}$ must either contain an entry with a hash value of the image (with a supported hash type), or it must contain an entry with a certificate against which an entry in the image’s certificate table can be verified. In either case verification must not succeed if the security database $\text{dbx}$ contains any record with:

A. Any entry with $\text{SignatureListType}$ of $\text{EFI\_CERT\_SHA256\_GUID}$ with any $\text{SignatureData}$ containing the SHA-256 hash of the binary.

B. Any entry with $\text{SignatureListType}$ of $\text{EFI\_CERT\_X509\_SHA256}$, $\text{EFI\_CERT\_X509\_SHA384}$, or $\text{EFI\_CERT\_X509\_SHA512}$, with any $\text{SignatureData}$ which reflects the To-Be-Signed hash included in any certificate in the signing chain of the signature being verified.

C. Any entry with $\text{SignatureListType}$ of $\text{EFI\_CERT\_X509\_GUID}$, with $\text{SignatureData}$ which contains a certificate with the same Issuer, Serial Number, and To-Be-Signed hash included in any certificate in the signing chain of the signature being verified.

Multiple signatures are allowed to exist in the binary’s certificate table (as per PE/COFF Section “Attribute Certificate Table”). Only one hash or signature is required to be present in $\text{db}$ in order to pass validation, so long as neither the SHA-256 hash of the binary nor any present signature is reflected in $\text{dbx}$.

Then, based on this match or its own policy, the firmware can decide whether or not to launch the UEFI image.
4. Start UEFI Image. If the UEFI Image is approved, then it is launched normally.
5. UEFI Image Not Approved. If the UEFI image was not approved the platform firmware may use other methods to discover if the UEFI image is authorized, such as consult a disk-based catalog or ask an authorized user. The result can be one of three responses: Yes, No or Defer.
6. UEFI Image Signature Added To Signature Database. If the user approves of the UEFI image, then the UEFI image’s signature is saved in the firmware’s signature database. If user approval is supported, then the firmware be able to update of the Signature Database. For more information, see Signature Database Update.
7. Go To Next Boot Option. If an UEFI image is rejected, then the next boot option is selected normally and go to step 3. This is in the case where the image is listed as a boot option.
8. UEFI Image Signature Passed In System Configuration Table. If user defers, then the UEFI image signature is copied into the Image Execution Information Table in the EFI System Configuration Table which is available to the operating system.
9. OS Application Validates UEFI Image. An OS application determines whether the image is valid.
10. UEFI Image Signature Added To Signature Database. For more information, see Signature Database Update.
11. End.
32.6 Code Definitions

32.6.1 UEFI Image Variable GUID & Variable Name

Summary
Constants used for UEFI signature database variable access.

Prototype

```c
#define EFI_IMAGE_SECURITY_DATABASE_GUID \
 { 0xd719b2cb, 0x3d3a, 0x4596, \
 { 0xa3, 0xbc, 0xda, 0xd0, 0x0e, 0x67, 0x65, 0x6f }}
#define EFI_IMAGE_SECURITY_DATABASE L"db"
#define EFI_IMAGE_SECURITY_DATABASE1 L"dbx"
#define EFI_IMAGE_SECURITY_DATABASE2 L"dbt"
#define EFI_IMAGE_SECURITY_DATABASE3 L"dbr"
```

Description

- This GUID and name are used when calling the EFI Runtime Services `GetVariable()` and `SetVariable()`.
- The `EFI_IMAGE_SECURITY_DATABASE_GUID` and `EFI_IMAGE_SECURITY_DATABASE` are used to retrieve and change the authorized signature database.
- The `EFI_IMAGE_SECURITY_DATABASE_GUID` and `EFI_IMAGE_SECURITY_DATABASE1` are used to retrieve and change the forbidden signature database.
- The `EFI_IMAGE_SECURITY_DATABASE_GUID` and `EFI_IMAGE_SECURITY_DATABASE2` are used to retrieve and change the authorized timestamp signature database.
- The `EFI_IMAGE_SECURITY_DATABASE_GUID` and `EFI_IMAGE_SECURITY_DATABASE3` are used to retrieve and change the authorized recovery signature database.
- Firmware shall support the `EFI_VARIABLE_APPEND_WRITE` flag (see Section 8.1.1) for the UEFI signature database variables.
- The signature database variables db, dbt, dbx, and dbr must be stored in tamper-resistant non-volatile storage.
33 - Human Interface Infrastructure Overview

This section defines the core code and services that are required for an implementation of the Human Interface Infrastructure (HII). This specification does the following:

- Describes the basic mechanisms to manage user input
- Provides code definitions for the HII-related protocols, functions, and type definitions that are architecturally required by the UEFI Specification

33.1 Goals

This chapter describes the mechanisms by which UEFI-compliant systems manage user input. The major areas described include the following:

- String and font management.
- User input abstractions (for keyboards and mice)
- Internal representations of the forms (in the HTML sense) that are used for running a preboot setup.
- External representations (and derivations) of the forms that are used to pass configuration information to runtime applications, and the mechanisms to allow the results of those applications to be driven back into the firmware.

General goals include:

- Simplified localization, the process by which the interface is adapted to a particular language.
- A "forms" representation mechanism that is rich enough to support the complex configuration issues encountered by platform developers, including stock keeping unit (SKU) management and interrelationships between questions in the forms.
- Definition of a mechanism to allow most or all the configuration of the system to be performed during boot, at runtime, and remotely. Where possible, the forms describing the configuration should be expressed using existing standards such as XML.
- Ability for the different drivers (including those from add-in cards) and applications to contribute forms, strings, and fonts in a uniform manner while still allowing innovation in the look and feel for Setup.

Support user-interface on a wide range of display devices:

- Local text display
- Local graphics display
- Remote text display
- Remote graphics display
- Web browser
- OS-present GUI

Support automated configuration without a display.
33.2 Design Discussion

This section describes the basic concepts behind the Human Interface Infrastructure. This is a set of protocols that allow a UEFI driver to provide the ability to register user interface and configuration content with the platform firmware. Unlike legacy option ROMs, the configuration of drivers and controllers is delayed until a platform management utility chooses to use the services of these protocols. UEFI drivers are not allowed to perform setup-like operations outside the context of these protocols. This means that a driver is not allowed to interact with the user outside the context of this protocol.

The following example shows a basic platform configuration or “setup” model. The drivers and applications install elements (such as fonts, strings, images and forms) into the HII Database, which acts as a central repository for the entire platform. The Forms Browser uses these elements to render the user interface on the display devices and receive information from the user via HID devices. When complete, the changes made by the user in the Forms Browser are saved, either to the UEFI global variable storage—(GetVariable() and SetVariable())—or to variable storage provided by the individual drivers.

---

**Figure 33-1 Platform Configuration Overview**

33.2.1 Drivers And Applications

The user interface elements in the form of package lists are carried by the drivers and applications. Drivers and applications can create the package lists dynamically, or they can be pre-built and carried as resources in the driver/application image.

If they are stored as resources, then an editor can be used to modify the user interface elements without recompiling. For example, display elements can be modified or deleted, new languages added, and default values modified.
The means by which the string, font, image and form resources are created is beyond the scope of this specification. The following diagram shows a few possible implementations. In both cases, the GUI design is an optional element and the user-interface elements are stored within a text-based resource file. Eventually, this source file is converted into a RES file (PE/COFF Resource Section) which can be linked with the main application.
Figure 33-3 Creating UI Resources With Resource Files
33.2.1.1 Platform and Driver Configuration

The intent is for this specification to enable the configuration of various target components in the system. The normally arduous task of managing user interface and configuration can be greatly simplified for the consumers of such functionality by enabling the platform to comprehend some standard user interactions.
33.2.1.2 Pre-O/S applications

There are various scenarios where a platform component must interact in some fashion with the user. Examples of this are when presenting a user with several choices of information (e.g. boot menu) and sending information to the display (e.g. system status, logo, etc.).
33.2.1.3 Description of User Interface Components

Various components listed in this specification are described in greater detail in their own sections. The user interface is composed of several distinct components illustrated below.

![User Interface Components Diagram](image)

**Figure 33-7 User Interface Components**

33.2.1.4 Forms

This component describes what type of content needs to be displayed to the user by means of a binary encoding (i.e., Internal Forms Representation) and also has added context information such as how to validate certain input and further describes where to store such input if it is intended to be non-volatile. Applications such as a browser or script engine may use the information with the forms to validate configuration setting values with or without a user interface.

33.2.1.5 Strings

The strings are the text-based (UCS-2 encoded) representations of the information typically being referenced by the forms. The intent of this infrastructure is also to seamlessly enable multiple language support. To that end the strings have the appropriate language designators to differentiate one language from another.

33.2.1.6 Images/Fonts

Since most content is typically intended to have the ability to be rendered on the local system, the human interface infrastructure also supports the ability for images and fonts to be accepted and used by the underlying user interface components.

33.2.1.7 Consumers of the user interface data

The ultimate consumer of the user interface information will be some type of forms browser or forms processor. There are several usage scenarios which should be supported by this specification. These are illustrated below:
33.2.1.8 Connected forms browser/processor

The ability to have the forms processing engine render content when directly connected to the target platform should be apparent. From the forms processing engine perspective, this could be the local machine or a machine that is network attached. In either case, there is a constructed agent which feeds the material to the forms processor for purposes of rendering the user interface and interacting with the user. Note that a forms processor could simply act on the forms data without ever having to render the user interface and interact with the user. This situation is much more akin to script processing and should be a very supportable situation.

---

33.2.1.9 Disconnected Forms Browser/Processor

By enabling the ability to import and export a platform’s settings, this infrastructure can also enable the ability for offline configuration. In this instance, a forms processor can interpret a given platform’s form data and enable (either through user interaction or through automated scripting) the changing of configuration settings. These settings can then be applied to the target platform when a connection is established.
33.2.1.10 O/S-Present Forms Browser/Processor
When it is desired that the forms data be used in the presence of an O/S, this specification describes a means by which to support this capability. By being able to encapsulate the data and export it through standard means such that an O/S agent (e.g. forms browser/processor) can retrieve it, O/S-present usage models can be made available for further value-add implementations.

Figure 33-10 O/S-Present Forms Browser/Processor

33.2.1.11 Where are the Results Stored
The forms data encodes how to store the changes per configuration question. The ability to save data to the platform as well as to a proprietary on-board store is provided. The premise is that each of the target non-volatile store components (e.g. motherboard, add-in device, etc.) would advertise an interface as described in this specification so that the forms browser/processor can route changes to the appropriate target.
33.2.2 Localization

Localization is the process by which the interface is adapted to a particular language. The table below discusses issues with localization and provides possible solutions.

Table 33-1 Localization Issues

<table>
<thead>
<tr>
<th>Issue</th>
<th>Example</th>
<th>Solution</th>
<th>Comment</th>
</tr>
</thead>
<tbody>
<tr>
<td>Directional display</td>
<td>Right to left printing for Hebrew.</td>
<td>Printing direction is a function of the language.</td>
<td>The display engine may or may not support all display techniques. If a language supports a display mechanism that the display engine does not, the language that uses the font must be selected.</td>
</tr>
<tr>
<td>Punctuation</td>
<td>Punctuation is directional. A comma in a right-to-left language is different from a comma in a left-to-right language.</td>
<td>Character choice is the choice of the author or translator.</td>
<td></td>
</tr>
<tr>
<td>Line breakage</td>
<td>Rules vary from language to language.</td>
<td>The UEFI preboot GUI performs little or no formatting.</td>
<td>The runtime display depends on the runtime browser and is not defined here.</td>
</tr>
</tbody>
</table>
33.2.3 User Input

To limit the number of required glyphs, we must also limit the amount and type of user input.

User input generally comes from the following main types of devices:

- Keyboards
- Mouse-like pointing devices

Input from other devices, such as limited keys on a front panel, can be handled two ways:

- Treat the limited keys as special-purpose devices with completely unique interfaces.
- Programmatically make the limited keys mimic a keyboard or mouse-like pointing device.

Pointing devices require no localization. They are universally understood by the subset of the world population addressed in this specification. For example, if a person does not know how to use a mouse or other pointing device, it is probably not a good idea to allow that person to change a system’s configuration.

On the other hand, keyboards are localized at the keycaps but not in the electronics. In other words, a French keyboard and a German keyboard might have very different keys but the software inside the keyboard—let alone the software in the system at the other end of the wire—cannot know which set of keycaps are installed.

This specification proposes to solve this issue by using the keys that are common between keyboards and ignoring language-specific keys. Keys that are available on USB keyboards in preboot mode include the following:

- Function keys (F1 – F12)
- Number keys (0-9)
- "Upside down T" cursor keys (the arrows, home, end, page up, page down)
- Numeric keypad keys
- The Enter, Space, Tab, and Esc keys
- Modifier keys (shifts, alts, controls, Windows*)

<table>
<thead>
<tr>
<th>Issue</th>
<th>Example</th>
<th>Solution</th>
<th>Comment</th>
</tr>
</thead>
<tbody>
<tr>
<td>Date and time</td>
<td>Most Europeans would write July 4, 1776, as 4/7/1776 while the United States would write it 7/4/1776 and others would write 1776/7/4. The separator characters between the parts of both date and time vary as well.</td>
<td>Generally left to the creator of the user interface.</td>
<td></td>
</tr>
<tr>
<td>Numbers</td>
<td>12,345.67 in one language is presented as 12.345,67 in another.</td>
<td>Print only integers and do not insert separator characters.</td>
<td>This solution is gaining acceptance around the world as more people use computers.</td>
</tr>
</tbody>
</table>
Number lock

The scan codes for these keys do not vary from language to language. These keys are the standard keys used for browser navigation although most end-users are unaware of this fact. Help for form-entry-specific keys must be provided to enable a useful keys-only interface. The one case where other, language-specific keys may be used is to enter passwords. Because passwords are never displayed, there is no requirement to translate scan code to Unicode character codes (keyboard localization) or scan codes to font glyphs.

Additional data can be provided to enable a richer set of input characters. This input is necessary to support features such as arbitrary text input and passwords.

33.2.4 Keyboard Layout

33.2.4.1 Keyboard Mapping

UEFI’s keyboard mapping loosely based definitions on ISO 9995. It bases the naming mechanism on the figure below. The keys highlighted in brown are the keys that nearly all keyboard layouts use for customizations. However, customization does not necessarily mean that all the keys are different. In fact, most of the keys are likely to be the same. When modifying the mapping, one can normally reference the keys in brown as the likely candidates (for whom to create modifications).

![Figure 33-12 Keyboard Layout](image)

Instead of referencing keys in hardware-specific ways such as scan codes, the HII specification defines an `EFI_KEY` enumeration that allows for a simple method of referencing this hardware abstraction. Type `EFI_KEY` is defined in `EFI_HII_DATABASE_PROTOCOL.GetKeyboardLayout()`. It also provides a way to update the keyboard layout with a great deal of flexibility. Any of the keys can be mapped to any 16-bit Unicode character code or control code value.

When defining the values for a particular key, there are six elements that are pertinent to the key:

- **Key name**
  - The `EFI_KEY` enumeration defines the names of the above keys.

- **Unicode Character Code**
  - Defines the Unicode Character Code (if any) of the named key.

- **Shifted Unicode Character Code**
  - Defines the Unicode Character Code (if any) of the named key while the shift modifier key is being pressed.
Alt-GR Unicode Character Code
Defines the Unicode Character Code (if any) of the named key while the Alt-GR modifier key (if any) is being pressed.

Shifted Alt-GR Unicode Character Code
Defines the Unicode Character Code (if any) of the named key while the Shift and Alt-GR modifier key (if any) is being pressed.

Modifier key value
Defines the nonprintable special function that this key has assigned to it.
- Under normal circumstances, a key that has any Unicode character code definitions generally has a modifier key value of EFI_NULL_MODIFIER. This value means the key has no special function other than the printing of a character. An exception to the rule is if any of the Unicode character codes have a value of 0xFFFF. Although rarely used, this value is the one case in which a key might have both a printable character and an active control key value.

An example of this exception would be the numeric keypad’s insert key. The definition for this key on a standard US keyboard is as follows:

<table>
<thead>
<tr>
<th>Key</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>EfiKeyZero</td>
<td></td>
</tr>
<tr>
<td>Unicode</td>
<td>0x0030 (basically ‘0’)</td>
</tr>
<tr>
<td>ShiftedUnicode</td>
<td>0xFFF (the exception to the rule)</td>
</tr>
<tr>
<td>AltGrUnicode</td>
<td>0x0000</td>
</tr>
<tr>
<td>ShiftedAltGrUnicode</td>
<td>0x0000</td>
</tr>
<tr>
<td>Modifier</td>
<td>EFI_INSERT_MODIFIER</td>
</tr>
</tbody>
</table>

This key is one of the few keys that, under normal circumstances, prints something out but also has a special function. These special functions are generally limited to the numeric keypad; however, this general limitation does not prevent someone from having the flexibility of defining these types of variations.

33.2.4.2 Modifier Keys
The definitions of the modifier keys allow for special functionality that is not necessarily accomplished by a printable character. Many of these modifier keys are flags to toggle certain state bits on and off inside of a keyboard driver. An example is EFI_CAPS_LOCK_MODIFIER. This state being active could alter what the typing of a particular key produces. Other control keys, such as EFI_LEFT_ARROW_MODIFIER and EFI_END_MODIFIER, affect the position of the cursor. One modifier key is likely unfamiliar to most people who exclusively use US keyboards, and that key is the EFI_ALT_GR_MODIFIER key. This key’s primary purpose is to activate a secondary type of shift modifier that exposes additional printable characters on certain keys. In some keyboard layouts, this key does not exist and is normally the EFI_RIGHT_ALT_MODIFIER key. None of the other modifier key functions should be a mystery to someone familiar with the usage of a standard computer keyboard.

An example of a few descriptor entries would be as follows:
Layout = {
    EfiKeyLCtrl, 0, 0, 0, 0, EFI_LEFT_CONTROL_MODIFIER, // Left control
    // key
    EfiKeyA0, 0, 0, 0, 0, EFI_NULL_MODIFIER,      // Not defined
    // Windows key
    EfiKeySpaceBar, 0x0020, 0x0020, 0x0020, 0x0020, EFI_NULL_MODIFIER
    // (Space Bar)
}

See "Related Definitions" in EFI_HII_DATABASE_PROTOCOL.GetKeyboardLayout() for the defined modifier values.

### 33.2.4.3 Non-spacing Keys

Non-spacing keys are a concept that provides the ability to OR together an accent key and another printable character. Non-spacing keys are defined as special types of modifier characters. They are typically accent keys that do not advance the cursor and in essence are a type of modifier key in that they maintain some level of state.

The way a person uses a non-spacing key is that the non-spacing key that maybe has the function of overlaying an umlaut (two dots) onto whatever the next character might be. The user presses the umlaut non-spacing key and follows it with a capital A, which yields an "Ä."

An example of a few descriptor entries would be as follows:

```c
    // If it's a dead key, we need to pass a list of physical key
    // names, each with a unicode, shifted, altgr, shiftedaltgr
    // character code. Each key name will have a Modifier value of
    // EFI_NS_KEY_MODIFIER for the first entry, and then the list of
    // EFI_NS_KEY_DEPENDENCY_MODIFIER physical key descriptions.
    // This eventually will lead to the next normal non-modifier key
    // definition.
    //
    // This requires defining an additional Modifier value of
    // EFI_NS_KEY_DEPENDENCY_MODIFIER to signify
    // EFI_NS_KEY_MODIFIER children definitions.
    //
    // The keyboard driver (consumer of the layouts) will know that
    // any key definitions with the EFI_NS_KEY_DEPENDENCY_MODIFIER
    // modifier do not redefine the value of the specified EFI_KEY.
    // They are simply used as a special case augmentation to the
    // original EFI_NS_KEY_MODIFIER.
    //
    // It is an error condition to define a
    // EFI_NS_KEY_MODIFIER without having all the
    // EFI_NS_KEY_DEPENDENCY_MODIFIER keys defined serially.
    //
    Layout = {
        EfiKeyE0, 0, 0, 0, 0, EFI_NS_KEY_MODIFIER,
```
In the above example, a key located at E0 is designated as a dead key. Using a common German keyboard layout as the example, a circumflex accent "^" is defined as a dead key at the E0 location. The A, E, I, O, and U characters are valid keys that can be pressed after the dead key and will produce a valid printable character. These characters are located at C1, D3, D8, D9, and D7 respectively.

The results of the Layout definition provided above would allow for the production of the following characters: âˆšå™©ëŒšø–º.

### 33.2.5 Forms

This specification describes how a UEFI driver or application may present a forms (or dialogs) based interface. The forms-based interface assumes that each window or screen consists of some window dressing (title & buttons) and a list of questions. These questions represent individual configuration settings for the application or driver, although several GUI controls may be used for one question.

---

**Figure 33-13 Forms-based Interface Example**

The forms are stored in the HII database, along with the strings, fonts and images. The various attributes of the forms and questions are encoded in IFR (Internal Forms Representation)—with each object and attribute a byte stream.

Other applications (so-called “Forms Processors”) may use the information within the forms to validate configuration setting values without a user interface at all.

The Forms Browser provides a forms-based user interface which understands how to read the contents of the forms, interact with the user, and save the resulting values. The Forms Browser uses forms data installed by an application or driver during initialization in the HII database. The Forms Browser organizes the forms so that a user may navigate between the forms, select the individual questions and change the values using the HID and display devices. When the user has finished making modifications, the Forms...
Browser saves the values, either to the global EFI variable store or else to a private variable store provided by the driver or application.

---

**Figure 33-14 Platform Configuration Overview**

### 33.2.5.1 Form Sets

*Form sets* are logically-related groups of forms.

**Attributes**

Each forms set has the following attributes:

**Form Set Identifier**

Uniquely identifies the form set within a package list using a GUID. The Form Set Identifier, along with a device path, uniquely identifies a form set in a system.

**Form Set Class Identifier**

Optional array of up to three GUIDs which identify how the form set should be used or classified. The list of standard form set classes is found in the "Related Definitions" section of **EFI_FORM_BROWSER2_PROTOCOL.SendForm()**.

**Title**

Title text for the form set.

**Help**

Help text for the form set.
Image
Optional title image for the form set.

Animation
Optional title animation for the form set

Description
Within a form set, there is one parent form and zero or more child forms. The parent form is the first enabled, visible form in the form set. The child forms are the second or later enabled, visible forms in the form set. In general, the Forms Browser will provide a means to navigate to the parent form. A cross-reference (see Section 33.2.5.4.10) is used to navigate between forms within a form set or between forms in different form sets.

Variable stores are declared within a form set. Variable stores describe the means for retrieval and storage of configuration settings, and location information within that variable store. For more information, see Section 33.2.5.6.

Default stores are declared within a form set. Default stores group together different types of default settings (normal, manufacturing, etc.) and give them a name. See Section 33.2.5.8 for more information.

The form set can control whether or not to process an individual form by nesting it inside of an EFI_IFR_DISABLE_IF expression. See Section 33.2.5.2.1 for more information. The form set can control whether or not to display an individual form by nesting it inside of an EFI_IFR_SUPPRESS_IF expression.

Syntax
The form set consists of an EFI_IFR_FORM_SET object, where the body consists of

```
form-set := EFI_IFR_FORM_SET form-set-list
```

33.2.5.2 Forms
Forms are logically-related groups of statements (including questions) designed to be displayed together.
Attributes
Each form has the following attributes:

Form Identifier
A 16-bit unsigned integer, which uniquely identifies the form within the form set.
The Form Identifier, along with the device path and Form Set Identifier, uniquely
identifies a form within a system.

Title
Title text for the form. The Forms Browser may use this text to describe the nature
and purpose of the form in a window title.

Image
Optional title image for the form. The Forms Browser may use this image to display
the nature and purpose of the form in a window title.

Animation
Optional title animation for the form set.

Modal
If a form is modal, then the on-form interaction must be completed prior to
navigating to another form. See "User Interaction", Section 33.2.10.1.

The form can control whether or not to process a statement by nesting it inside of an
EFI_IFR_DISABLE_IF expression. See Section 33.2.5.3.2 for more information.

The form can control whether a particular statement is selectable by nesting it inside of an
EFI_IFR_GRAY_OUT_IF expression. Statements that cannot be selected are displayed by Form
Browsers, but cannot be selected by a user. EFI_IFR_GRAY_OUT_IF causes statements to be displayed
with some visual indication. See Section 33.2.5.3.4 for more information.

The form can control whether to display a statement by nesting it inside of an EFI_IFR_SUPPRESS_IF
expression. See Section 33.3.8.3.75 for more information.

Syntax
The form consists of an EFI_IFR_FORM object, where the body consists of:

form:=EFI_IFR_FORM form-tag-list |
EFI_IFR_FORM_MAP form-tag-list
form-tag-list:=form-tag form-tag-list |
<empty>
form-tag:=EFI_IFR_IMAGE |
EFI_IFR_ANIMATION |
EFI_IFR_LOCKED |
EFI_IFR_RULE |
 EFI_IFR_MODAL_TAG | 
statement | 
question | 
cond-statement-list | 
<empty>

statement-list:= statement statement-list | 
question statement-list | 
cond-statement-list | 
<empty>

cond-statement-list:= EFI_IFR_DISABLE_IF expression statement-list | 
EFI_IFR_SUPPRESS_IF expression statement-list | 
EFI_IFR_GRAY_OUT_IF expression statement-list | 
question-list := question question-list | 
<empty>

Other unknown opcodes are permitted, but will be ignored.

33.2.5.2.1 Enable/Disable
Disabled forms will not be processed at all by a Forms Processor. Forms are enabled unless:

- The form nests inside an EFI_IFR_DISABLE_IF expression which evaluated to false.
- The disabling of forms is evaluated during Forms Processor initialization and is not re-evaluated.

33.2.5.2.2 Modifiability
Forms can be locked so that a Forms Editor will not change it. Forms are unlocked unless:

- The form has an EFI_IFR_LOCKED in its scope.

The locking of statement is evaluated only during Forms Editor initialization.

33.2.5.2.3 Visibility
Suppressed forms will not be displayed. Forms are visible unless:

- The form is disabled (see Section 33.2.5.4)
- The form is nested inside an EFI_IFR_SUPPRESS_IF expression which evaluates to false.
33.2.5.3 Statements

All displayable items within the body of a form are statements. Statements provide information or capabilities to the user. Questions (see Section 33.2.5.4) are a specialized form of statement with a value. Statements are used only by Forms Browsers and are ignored by other Forms Processors.

Attributes

Statements have the following attributes:

- **Prompt**
  The text that will be displayed with the statement.

- **Help**
  The extended descriptive text that can be displayed with the statement.

- **Image**
  The optional image that will be displayed with the statement.

- **Animation**
  The optional animation that will be displayed with the statement.

Other than Questions, there are three types of statements:

- Static Text/Image
- Subtitle
- Cross-Reference

Syntax

```
statement ::= subtitle | static-text | reset button
statement-tag-list ::= statement-tag statement-tag-list | <empty>
statement-tag ::= EFI_IFR_IMAGE | EFI_IFR_LOCKED | EFI_IFR_ANIMATION
```

33.2.5.3.1 Display

Statement display depends on the Forms Browser. Statements do not describe how the statement must be displayed but rather provide resources (such as text and images) for use by the Forms Browser. The Forms Browser uses this information to create the necessary user interface.

The Forms Browser may use the visibility (see Section 33.2.5.3) or selectability (see Section 33.2.5.4) of the statements to change the way the item is displayed. The `EFI_IFR_GRAY_OUT_IF` expression explicitly requires that nested statements have visual differentiation from normal statements.
### 33.2.5.3.2 Enable/Disable

Statements which have been disabled will not be processed at all by a Forms Processor. Statements are enabled unless:

- The parent statement or question is disabled.
- The statement is nested inside an `EFI_IFR_DISABLE_IF` expression which evaluated to false.
- The disabling of statements is evaluated during Forms Browser initialization and is not re- evaluated.

### 33.2.5.3.3 Visibility

Suppressed statements will not be displayed. Statements are displayed unless:

- The parent statement or question is suppressed.
- The statement is disabled (see Section 33.2.5.3.2).
- The statement is nested inside an `EFI_IFR_SUPPRESS_IF` expression which evaluated to false.

The suppression of the statements is evaluated during Forms Browser initialization. Subsequently, the suppression of statements is reevaluated each time a value in any question on the selected form has changed.

### 33.2.5.3.4 Evaluation of Selectable Statements

A user in a Forms Browser can choose statements which are selectable. Statements are selectable unless:

- The parent statement or question is not selectable.
- The statement is suppressed (see Section 33.2.5.3.2).
- The statement is nested inside an `EFI_IFR_GRAY_OUT_IF` expression which evaluated to false.

The evaluation of selectable statements takes place during Forms Browser initialization. Subsequently, selectable statements are reevaluated each time a value in any question on the selected form has changed.

### 33.2.5.3.5 Modifiability

A statement can be locked so that a Forms Editor will not change it. Statements are unlocked unless:

- The parent form or parent statement/question is locked.
- The statement has an `EFI_IFR_LOCKED` in its scope.

The locking of a statement is evaluated only during Forms Editor initialization.

### 33.2.5.3.6 Static Text/Image

The Forms Browser displays the specified prompt, the specified text and (optionally) the image, but has no user interaction.

**Syntax**

```
static-text:= EFI_IFR_TEXT  statement-tag-list
```
33.2.5.3.7 Subtitle
The subtitle is a means of visually grouping questions by providing a separator, some optional separating text, and an optional image.

Syntax
subtitle := EFI_IFR_SUBTITLE statement-tag-list

33.2.5.3.8 Reset Button
Attributes
Reset Buttons have the following attributes:

  Default Id
  Specifies the default set to use when restoring defaults to the current form.

Syntax
reset button := EFI_IFR_RESET_BUTTON statement-tag-list

33.2.5.4 Questions
Questions are statements which have a value. The value corresponds to a configuration setting for the platform or for a device. The question uniquely identifies the configuration setting, describes the possible values, the way the value is stored, and how the question should be displayed.

Attributes
Questions have the following attributes (in addition to those of statements):

  Question Identifier
  A 16-bit unsigned integer which uniquely identifies the question within the form set in which it appears. The Question Identifier, along with the device path and Form Set Identifier, uniquely identifies a question within a system.

  Default Value
  The value used when the user requests that defaults be loaded.

  Manufacturing Value
  The value used when the user requests that manufacturing defaults are loaded.

  Value
  Each question has a current value. See Section 33.2.5.4.1 for more information.

  Value Format
  The format used to store a question’s value.

  Value Storage
  The means by which values are stored. See Section 33.2.5.4.2 for more information.
Refresh Identifiers

Zero or more GUIDs associated with an event group initialized by the Forms Browser when the form set containing the question is opened. If the event group associated with the GUID is signaled (see `SignalEvent()`), then the question value will be updated from storage.

Refresh Interval

The minimum number of seconds that must pass before the Forms Browser will automatically update the current question value from storage. The default value is zero, indicating there will be no automatic refresh.

Validation

New values assigned to questions can be validated, using validation expressions, or, if connected, using a callback. See Section 33.2.5.9 for more information.

Callback

If set, the callback will be called when the question’s value is changed. In some cases, the presence of these callbacks prevents the question’s value from being edited while disconnected.

The question can control whether a particular option can be displayed by nesting it inside of an `EFI_IFR_SUPPRESS_IF` expression. Form Browsers do not display Suppressed Options, but Suppressed Options may still be examined by Form Processors.

Syntax

```
question ::= action-button | boolean | date | number | ordered-list | string | time |
<empty>
question-tag ::= statement-tag |
EFI_IFR_INCONSISTENT_IF expression |
EFI_IFR_NO_SUBMIT_IF   expression |
EFI_IFR_WARNING_IF     expression |
EFI_IFR_DISABLE_IF     expression question-list |
EFI_IFR_REFRESH_ID    RefreshEventGroupId |
EFI_IFR_REFRESH       |
EFI_IFR_VARSTORE_DEVICE

question-option-tag ::= EFI_IFR_SUPPRESS_IF   expression |
EFI_IFR_VALUE         optional-expression |
```
33.2.5.4.1 Values

Question values are a data type listed in Section 33.2.5.7.4. During initialization of the Forms Processor or Forms Browser, the values of all enabled questions are retrieved. If the value cannot be retrieved, then the question’s value is **Undefined**.

A question with the value of type **Undefined** will be suppressed. This suppression will be reevaluated based on Value Refresh or when any question value on the selected form is changed.

When the form is submitted, the modified values are written to Value Storage. When the form is reset, the question value is set to the default question value. If there is no default question value, the question value is unchanged.

When a question value is retrieved, the following process is used:

1. Set the *this* internal constant to have the same value as the one read from the question’s storage.
2. If present, change the current question value to the value returned by a question’s nested **EFI_IFR_READ** operator.
When a question value is changed, the following process is used:

1. Set the `this` internal constant to have the same value as the current question value.
2. If present, evaluate the question’s nested `EFI_IFR_WRITE` (Section 33.3.8.3.94) operator.
3. Write the value to the question’s storage.
33.2.5.4.2 Storage Requirements

Question storage requirements describe the type and size of storage for the value. These storage requirements describe whether the question’s value will be stored as an EFI global variable or using driver local storage. It also describes whether the value is packed together with other values in a buffer, or passed as a name-value pair. See Section 33.2.5.6 for more information.

33.2.5.4.3 Display

Question display depends on the Forms Browser. Questions do not describe how the question must be displayed. Instead, questions provide resources (such as text and images) and information about visibility and the ability to edit the question. The Forms Browser uses these to create the necessary user interface.

Questions can have prompt text, help text and (optionally) an image. The prompt text usually describes the nature of the question. Help text is displayed either in a special display area or only at the request of the user. Questions can also have hints which describe how to visually organize the information.

33.2.5.4.4 Action Button

Action buttons are buttons which cause a pre-defined configuration string to process immediately. There is no storage directly associated with the button.
**Attributes**
Action buttons have no additional attributes other than the common question attributes).

**Storage**
There is no storage associated with the action button.

**Results**
There are no results associated with the action button. If used in an expression, the question value will always be **Undefined**.

**Syntax**

```plaintext
action-button:= EFI_IFR_ACTION question-tag-list
```

### 33.2.5.4.5 Boolean

Boolean questions are those that allow a choice between true and false. The question’s value is Boolean. In general, construct questions so that the prompt text asks questions resulting in ‘yes/enabled/on’ is ‘true’ and ‘no/disabled/off’ is ‘false’.

Boolean questions may be displayed as a check box, two radio buttons, a selection list, a list box, or a drop list box.

**Attributes**
Boolean questions have no additional attributes other than the common question attributes:

**Storage**
If the boolean question uses Buffer storage or EFI Variable (see Section 33.2.5.6), then the size is exactly one byte, with the FALSE condition is zero and the TRUE value is 1.

**Results**
The results are represented as either 0 (**FALSE**) or 1 (**TRUE**).

**Syntax**

```plaintext
boolean:= EFI_IFR_CHECKBOX question-option-list
```

### 33.2.5.4.6 Date

Date questions allow modification of part or all of a standard calendar date. The format of the date display depends on the Forms Browser and any localization.

**Attributes**
Date questions have the following attributes:

**Year Suppressed**
The year will not be displayed or updated.

**Month Suppressed**
The month will not be displayed or updated.
**Day Suppressed**

The day will not be displayed or updated.

**UEFI Storage**

In addition to normal question Value Storage, Date questions can optionally be instructed to save the date to either the system time or system wake-up time using the UEFI runtime services `SetTime()` or `SetWakeupTime()`. In this case, the date and time will be read first, the modifications made and changes will be written back.

Conversion to and from strings to a date depends on the system localization.

The date value is stored an `EFI_HII_TIME` structure. The TimeZone field is always set to `EFI_UNSPECIFIED_TIMEZONE`. The Daylight field is always set to zero. The contents of the other fields are undetermined.

**Storage**

If the date question uses Buffer storage or EFI Variable storage (see Section 33.2.5.6), then the stored result will occupy exactly the size of `EFI_HII_DATE`.

**Results**

Results for date questions are represented as a hex dump of the `EFI_HII_DATE` structure. If used in a question, the value will be a buffer containing the contents of the `EFI_HII_DATE` structure.

**Syntax**

\[ date \ := \ EFI_IFR_DATE \ \ question-option-list \]

**33.2.5.4.7 Number**

Number questions allow modification of an integer value up to 64-bits. Number questions can also specify pre-defined options.

**Attributes**

Number questions have the following attributes:

- **Radix**
  
  Hint describes the output radix of numbers. The possible values are unsigned decimal, signed decimal or hexadecimal. Numbers displayed in hexadecimal will be prefixed by ‘0x’

- **Minimum Value**

  The minimum unsigned value which can be accepted for this question.

- **Maximum Value**:

  The maximum unsigned value which can be accepted for this question.

- **Skip Value**:

  Defines the minimum increment between values.
Storage

If the number question uses Buffer storage or EFI Variable storage (see Section 33.2.5.6), then the buffer size specified by must be 1, 2, 4 or 8. Also, the Forms Processor will do implicit error checking to make sure that the signed or unsigned value can be stored in the Buffer without lost of significant bits. For example, if the buffer size is 1 byte, then the largest unsigned integer value would be 255. Likewise, the largest signed integer value would be 127 and the smallest signed integer value would be -128. The Forms Processor will automatically detect this as an error and generate an appropriate error.

Results

The results are represented as string versions of unsigned hexadecimal values.

Syntax

number := EFI_IFR_NUMERIC question-option-list |
          EFI_IFR_ONE_OF question-option-list

33.2.5.4.8 Set

Sets are questions where \( n \) containers can be filled with any of \( m \) pre-defined choices. This supports both lists where a given value can only appear in one of the slots or where the same choice can appear many times.

Each of the containers takes the form of an option which a name, a value and (optionally) an image.

Attributes

Set questions have the following attributes:

**Container Count**

Specifies the number of available selectable options.

**Unique**

If set, then each choice may be used at most, once.

**NoEmpty**

All slots must be filled with a non-zero value.

**Storage**

The set questions are stored as a Buffer with one byte for each Container.

Results

Each Container value is represented as two characters, one for each nibble. All hexadecimal characters (a-f) are in lower-case.

The results are represented as a series of Container values, starting with the lowest Container.

Syntax

ordered-list := EFI_IFR_ORDERED_LIST question-option-list
Options
Set questions treat the values specified by nested EFI_IFR_ONE_OF_OPTION values as the value for a single Container, not the entire question storage. This is different from other question types.

Defaults
Set questions treat the default values specified by nested EFI_IFR_DEFAULT or EFI_IFR_ONE_OF_OPTION opcodes as the default value for all Containers. The default values must be of type EFI_IFR_TYPE_BUFFER, with each byte in the buffer corresponding to a single Container value, starting with the first container. If the buffer contains fewer bytes than MaxContainers, then the remaining Containers will be set to a value of 0.

Default values returned from the ALTCFG section when ExtractConfig() is called fill the storage starting with the first container.

33.2.5.4.9 String
String questions allow modification of a string.

Attributes
String questions have the following attributes:

Minimum Length
Hint describes the minimum length of the string, in characters.

Maximum Length
Hint describes the maximum length of the string, in characters.

Multi-Line
Hint describes that the string might contain multiple lines.

Output Mask
If set, the text entered will not be displayed.

Storage
The string questions are stored as a NULL-terminated string. If the time question uses Buffer or EFI Variable storage (see Section 33.2.5.6), then the buffer size must exceed the size of the NULL-terminated string. If the string is shorter than the length of the buffer, the remainder of the buffer is filled with NULL characters.

Results
Results for string questions are represented as hex dump of the string, including the terminating NULL character.

Syntax
string := EFI_IFR_STRING question-option-list |
EFI_IFR_PASSWORD question-option-list
33.2.5.4.10 Cross-Reference

Cross-reference questions provide a selectable means by which users navigate to other forms and/or other questions. The form and question can be in the current form set, another form set or even in a form associated with a different device. If the specified form or question does not exist, the button is not selectable, is grayed-out, or is suppressed.

Attributes

Cross references can have the following attributes:

- **Form Identifier**
  - The identifier of the target form.

- **Form Set Identifier**
  - Optionally specifies an alternate form-set which contains the target form. If specified, then the focus will be on form within the form set specified by Form Identifier. If the Form Identifier is not specified, then the first form in the Form Set is used.

- **Question Identifier**
  - Optionally specifies the question identifier of the target question on the target form. If specified then focus will be placed on the question specified by this question identifier. Otherwise, the focus will be on the first question within the specified form.

- **Device Path**
  - Optionally, the device path which contains the Form Identifier. Otherwise, the device path associated with the form set containing this cross-reference will be used.

- **Storage**
  - Storage is optional for a cross-reference question. It is only present when the cross-reference question does not supply any target (i.e., REF5). If the question uses Buffer or EFI Variable storage (see Section 33.2.5.6), then the buffer size must be exactly the size of the **EFI_HII_REF** structure.

- **Results**
  - Results for cross-reference questions are represented as a hex dump of the question identifier, form identifier, form set GUID and null-terminated device path text. If used in a question, the question value will be a buffer containing the **EFI_HII_REF** structure..

**Syntax**

cross-reference := **EFI_IFR_REF**  statement-tag-list

33.2.5.4.11 Time

Time questions allow modification of part or all of a time. The format of the time display depends on the Forms Browser and any localization.
Attributes
Time questions have the following attributes:

**Hour Suppressed**
The hour will not be displayed or updated.

**Minute Suppressed**
The minute will not be displayed or updated.

**Second Suppressed**
The second will not be displayed or updated.

**UEFI Storage**
In addition to normal question Value Storage, time questions can be instructed to save the time to either the system time or system wake-up time using the UEFI runtime services `SetTime` or `SetWakeupTime`. In these instances, the date and time is read first, the modifications made and changes are then written back.

Conversion to and from strings to a time depends on the system localization.

The time value is stored as part of an `EFI_HII_TIME` structure. The contents of the other fields are undetermined.

**Storage**
If the time question uses Buffer or EFI Variable storage (see Section 33.2.5.6), then the buffer size must be exactly the size of the `EFI_HII_TIME` structure.

**Results**
Results for time questions are represented as a hex dump of the `EFI_HII_TIME` structure. If used in a question, the value will be a buffer containing the contents of the `EFI_HII_TIME` structure.

**Syntax**
```
time := EFI_IFR_TIME question-option-list
```

33.2.5.5 Options
Use Options within questions to give text or graphic description of a particular question value. They may also describe the choices in the set data type.

Attributes
Options have the following attributes:

**Text**
The text for the option.

**Image**
The optional image for the option.
Animation
The optional animation for the option.

Value
The value for the option.

Default
If set, this is the option selected when the user asks for the defaults. Only one visible option can have this bit set within a question’s scope.

Manufacturing Default
If set, this is the option selected when manufacturing defaults are set. Only one visible option can have this bit set within a question’s scope.

Syntax
```plaintext
option ::= EFI_IFR_ONE_OF_OPTION option-tag-list
option-tag-list ::= option-tag option-tag-list | <empty>
option-tag ::= EFI_IFR_IMAGE | EFI_IFR_ANIMATION
```

33.2.5.5.1 Visibility
Options which have been suppressed will not be displayed. Options are displayed unless:

- The parent question is suppressed.
- The option is nested inside an EFI_IFR_SUPPRESS_IF expression which evaluated to false.

The suppression of the options is evaluated each time the option is displayed.

33.2.5.6 Storage
Question values are stored in Variable Stores, which are application, platform or device repositories for configuration settings. In many cases, this is non-volatile storage. In other cases, it holds only the current behavior of a driver or application.

Question values are retrieved from the variable store when the form is initialized. They are updated periodically based on question settings and stored back in the variable store when the form is submitted.

It is possible for a question to have no associated Variable Store. This happens when the VarStoreID associated with the question is set to zero and, for Date/Time questions, the UEFI Storage is disabled. For questions with no associated Variable Store, the question must either support the RETRIEVE and CHANGED callback actions (see EFI_HII CONFIG ACCESS PROTOCOL.CallBack()) or contain an embedded READ or WRITE opcode: EFI_IFR_READ_OP and EFI_IFR_WRITE_OP (see Section 33.3.8.3.58 and Section 33.3.8.3.94).

Because the value associated with a question contained in a Variable Store can be shared by multiple questions, the questions must all treat the shared information as compatible data types. There are four types of variable stores:
Buffer Storage
With buffer storage, the application, platform or driver provides the definition of a buffer which contains the values for one or more questions. The size of the entire buffer is defined in the EFI_IFR_VARSTORE definition. Each question defines a field in the buffer by providing an offset within the buffer and the size of the required storage. These variable stores are exposed by the app/driver using the EFI_HII_CONFIG_ACCESS_PROTOCOL, which is installed on the same handle as the package list. Question values are retrieved via EFI_HII_CONFIG_ACCESS_PROTOCOL.ExtractConfig() and updated via EFI_HII_CONFIG_ACCESS_PROTOCOL.RouteConfig(). Rather than access the buffer as a whole, Buffer Storage Variable Stores access each field independently, via a list of one or more (field offset, value) pairs encoded as variable length text strings as defined for the EFI_HII_CONFIG_ACCESS_PROTOCOL.

Name/Value Storage
With name/value storage, the application provides a string which contains the encoded values for a single question. These variable stores are exposed by the app/driver using the EFI_HII_CONFIG_ACCESS_PROTOCOL, which is installed on the same handle as the package list.

EFI Variable Storage
This is a specialized form of Buffer Storage, which uses the EFI runtime services GetVariable() and SetVariable() to access the entire buffer defined for the Variable Store as a single binary object.

EFI Date/Time Storage
For date and time-related questions, the question values can be retrieved using the EFI runtime services GetTime() and GetWakeupTime() and stored using the EFI runtime services SetTime() and SetWakeupTime().

The following table summarizes the types of information needed for each type of storage and where it is retrieved from.

<table>
<thead>
<tr>
<th>Storage Type</th>
<th>Information Type</th>
<th>Where It Comes From</th>
</tr>
</thead>
<tbody>
<tr>
<td>None</td>
<td>Driver Handle</td>
<td>Handle specified with NewPackageList() or derived from EFI_IFR_VARSTORE_DEVICE.DevicePath</td>
</tr>
</tbody>
</table>
### 33.2.5.7 Expressions

This section describes the expressions used in various expressions in IFR. The expressions are encoded using normal IFR opcodes, but in RPN (Reverse Polish Notation) where the operands occur before the operator.

<table>
<thead>
<tr>
<th>Storage Type</th>
<th>Information Type</th>
<th>Where It Comes From</th>
</tr>
</thead>
<tbody>
<tr>
<td>Buffer Storage</td>
<td>Driver Handle</td>
<td>Handle specified with <code>NewPackageList()</code> or derived from <code>EFI_IFR_VARSTORE_DEVICE.DevicePath</code></td>
</tr>
<tr>
<td>Variable ID</td>
<td>Variable store specified by <code>EFI_IFR_QUESTION_HEADER.VarStoreId</code></td>
<td></td>
</tr>
<tr>
<td>Variable Name</td>
<td>Variable store specified by <code>EFI_IFR_QUESTION_HEADER.VarStoreId</code></td>
<td></td>
</tr>
<tr>
<td>Variable Store Offset</td>
<td>Variable store offset specified by <code>EFI_IFR_QUESTION_HEADER.VarOffset</code>.</td>
<td></td>
</tr>
<tr>
<td>Name/Value Storage</td>
<td>Driver Handle</td>
<td>Handle specified with <code>NewPackageList()</code> or derived from <code>EFI_IFR_VARSTORE_DEVICE.DevicePath</code></td>
</tr>
<tr>
<td>Variable ID</td>
<td>Variable store specified by <code>EFI_IFR_QUESTION_HEADER.VarStoreId</code></td>
<td></td>
</tr>
<tr>
<td>Variable Name</td>
<td>Variable name specified by <code>EFI_IFR_QUESTION_HEADER.VarStoreInfo.VarName</code>.</td>
<td></td>
</tr>
<tr>
<td>EFI Variable Storage</td>
<td>Driver Handle</td>
<td>None</td>
</tr>
<tr>
<td>Variable ID</td>
<td>Variable store specified by <code>EFI_IFR_QUESTION_HEADER.VarStoreId</code></td>
<td></td>
</tr>
<tr>
<td>EFI_Variable GUID (for Variable Services)</td>
<td>EFI variable GUID specified by <code>EFI_IFR_VARSTORE_EFI.Guid</code>.</td>
<td></td>
</tr>
<tr>
<td>EFI_Variable Name (for Variable Services)</td>
<td>EFI variable name specified by <code>EFI_IFR_VARSTORE_EFI.Name</code>.</td>
<td></td>
</tr>
<tr>
<td>Variable Name</td>
<td>Variable name specified by <code>EFI_IFR_QUESTION_HEADER.VarStoreId</code>.</td>
<td></td>
</tr>
<tr>
<td>Variable Store Offset</td>
<td>Variable store offset specified by <code>EFI_IFR_QUESTION_HEADER.VarStoreInfo.VarOffset</code>.</td>
<td></td>
</tr>
<tr>
<td>EFI Date/Time Storage</td>
<td>Driver Handle</td>
<td>None</td>
</tr>
<tr>
<td>Variable ID</td>
<td>None</td>
<td></td>
</tr>
<tr>
<td>Variable Name</td>
<td>None</td>
<td></td>
</tr>
</tbody>
</table>

---

**Example:**

```
DeviceHandle = NewPackageList()

VarStore = EFI_IFR_VARSTORE_DEVICE.DevicePath

VarID = EFI_IFR_QUESTION_HEADER.VarStoreId

VarName = EFI_IFR_QUESTION_HEADER.VarStoreInfo.VarName
```
The opcodes fall into these categories:

**Unary operators**
- Functions taking a single sub-expression.

**Binary operators.**
- Functions taking two sub-expressions.

**Ternary operators.**
- Functions taking three sub-expressions.

**Built-in functions.**
- Operators taking zero or more sub-expressions.

**Constants.**
- Numeric and string constants.

**Question Values.**
- Specified by their question identifier.

All integer operations are performed at 64-bit precision.

### 33.2.5.7.1 Expression Encoding

Expressions are usually encoded within the scope of another binary object. If the expression consists of more than a single opcode, the first opcode should open a scope (Header.Scope = 1) and use an **EFI_IFR_END** opcode to close the scope in order to make sure they can be skipped.

### 33.2.5.7.2 Expression Stack

When evaluating expressions, the Forms Processor uses a stack to hold intermediate values. Each operator either pushes a value on the stack, pops a value from the stack, or both. For example, the **EFI_IFR_ONE** operator pushes the integer value 1 on the expression stack. The **EFI_IFR_ADD** operator pops two integer values from the expression stack, adds them together, and pushes the result back on the stack.

After evaluating an expression, there should be only one value left on the expression stack.

### 33.2.5.7.3 Rules

Rules are pre-defined expressions attached to the form. These rules may be used in any expression within the form’s scope. Each rule is given a unique identifier (0-255) when it is created by **EFI_IFR_RULE**. This same identifier is used when the rule is referred to in an expression with **EFI_IFR_RULE_REF**.

To save space, rules are intended to allow manual or automatic extraction of common sub-expressions from form expressions.

### 33.2.5.7.4 Data Types

The expressions use five basic data types:

**Boolean**
- True or false.
**Unsigned Integer**

64-bit unsigned integer.

**String**

Null-terminated string.

**Buffer**

Fixed size array of unsigned 8-bit integers.

**Undefined**

Undetermined value. Used when the value cannot be calculated or for run-time errors.

Data conversion is not implicit. Explicit data conversion can be performed using the `EFI_IFR_TO_STRING`, `EFI_IFR_TO_UINT`, and `EFI_IFR_TO_BOOLEAN` operators.

The Date and Time question values are converted to the Buffer data type filled with the `EFI_HII_DATE` and `EFI_HII_TIME` structure contents (respectively).

The Ref question values are converted to the Buffer data type and filled with the `EFI_HII_REF` and structure contents.

**Syntax**

The expressions have the following syntax:

- `expression ::= built-in-function | constant | expression unary-op | expression expression binary-op | expression expression expression ternary-op`

**EFI_IFR_MAP**

- `expression-pair-list ::= expression-pair-list expression expression | <empty>`

- `optional-expression ::= expression | <empty>`

- `built-in-function ::= EFI_IFR_DUP | EFI_IFR_EQ_ID_VAL | EFI_IFR_EQ_ID_ID | EFI_IFR_EQ_ID_VAL_LIST |`
EFI_IFR_GET |
EFI_IFR_QUESTION_REF1 |
EFI_IFR_QUESTION_REF3 |
EFI_IFR_RULE_REF |
EFI_IFR_STRING_REF1 |
EFI_IFR_THIS |
EFI_IFR_SECURITY

constant := EFI_IFR_FALSE |
EFI_IFR_ONE |
EFI_IFR_ONES |
EFI_IFR_TRUE |
EFI_IFR_UINT8 |
EFI_IFR_UINT16 |
EFI_IFR_UINT32 |
EFI_IFR_UINT64 |
EFI_IFR_UNDEFINED |
EFI_IFR_VERSION |
EFI_IFR_ZERO

binary-op := EFI_IFR_ADD |
EFI_IFR_AND |
EFI_IFR_BITWISE_AND |
EFI_IFR_BITWISE_OR |
EFI_IFR_CATEGATE |
EFI_IFR_DIVIDE |
EFI_IFR_EQUAL |
EFI_IFR_GREATER_EQUAL |
EFI_IFR_GREATER_THAN |
EFI_IFR_LESS_EQUAL |
EFI_IFR_LESS_THAN |
EFI_IFR_MATCH |
EFI_IFR_MATCH2 |
33.2.5.8 Defaults

To ensure consistent behavior when a platform attempts to restore settings to defaults, each question op-code must have an active default setting. Defaults are pre-defined question values. The question values may be changed to their defaults either through a Forms Processor-defined means or when the user selects an `EFI_IFR_RESET_BUTTON` statement (see Section 33.2.5.3.8).

Each question may have zero or more default values, with each default value used for different purposes. For example, there might be a "standard" default value, a default value used for manufacturing and a "safe" default value. A group of default values used to configure a platform or device for a specific purpose is called default store.

Default Stores

There are three standard default stores:
Standard Defaults
These are the defaults used to prepare the system/device for normal operation.

Manufacturing Defaults
These are the defaults used to prepare the system/device for manufacturing.

Safe Defaults
These are the defaults used to boot the system in a “safe” or low-risk mode.

Attributes
Default stores have the following attributes:

Name
Each default store has a user-readable name

Identifier
A 16-bit unsigned integer. The values between 0x0000 and 0x3fff are reserved for use by the UEFI specification. The values between 0x4000 and 0x7fff are reserved for platform providers. The values between 0x8000 and 0xbfff are reserved for hardware vendors. The values between 0xc000 and 0xffff are reserved for firmware vendors.

```
#define EFI_HII_DEFAULT_CLASS_STANDARD     0x0000
#define EFI_HII_DEFAULT_CLASS_MANUFACTURING   0x0001
#define EFI_HII_DEFAULT_CLASS_SAFE       0x0002
#define EFI_HII_DEFAULT_CLASS_PLATFORM_BEGIN  0x4000
#define EFI_HII_DEFAULT_CLASS_PLATFORM_END   0x7fff
#define EFI_HII_DEFAULT_CLASS_HARDWARE_BEGIN  0x8000
#define EFI_HII_DEFAULT_CLASS_HARDWARE_END   0xbfff
#define EFI_HII_DEFAULT_CLASS_FIRMWARE_BEGIN  0xc000
#define EFI_HII_DEFAULT_CLASS_FIRMWARE_END   0xffff
```

Users of these ranges are encouraged to use the specification defined ranges for maximum interoperability. Questions or platforms may support defaults for only a sub-set of the possible default stores. Support for default store 0 (“standard”) is recommended.

Defaulting
When retrieving the default values for a question, the Forms Processor uses one of the following (listed from highest priority to lowest priority):

1. The value returned from the `Callback()` member function of the Config Access protocol associated with the question when called with the `Action` set to one of the `EFI_BROWSER_ACTION_DEFAULT_x` values (see Section 35.5). It is recommended that this form only be used for questions where the default value alters dynamically at runtime.

2. The value returned in the `Response` parameter of the `ConfigAccess()` member function (using the ALTCFG form). See Section 35.2.1.

3. The value specified by an `EFI_IFR_DEFAULT` opcodes appear within the scope of a question. (see Section 33.3.8.3.12)
4. One of the Options (see Section 33.2.5.5) has its Standard Default or Manufacturing Default attribute set.

5. For Boolean questions, the Standard Default or Manufacturing Default values in the Flags field. (see Section 33.2.5.4.5).

**Syntax**

```
Default := EFI_IFR_DEFAULT
default-tag := EFI_IFR_VALUE | <empty>
```

### 33.2.5.9 Validation

Validation is the process of determining whether a value can be applied to a configuration setting. Validation takes place at three different points in the editing process: edit-level, question-level and form-level.

#### 33.2.5.9.1 Edit-Level Validation

First, it takes place while the value is being edited with a Forms Browser. The Forms Browser may optionally reject values selected by the user which would fail Question-Level validation. For example, the Forms Browser may limit the length of strings entered so that they meet the Minimum and Maximum Length.

#### 33.2.5.9.2 Question-Level Validation

Second, it takes place when the value has changed, normally when the user attempts to leave the control, navigate between the portions of the control or selects one of the option values. At this point, an error occurs if:

- For a String (see Section 33.2.5.4.9), if the string length is less than the Minimum Length, then the Forms Processor generates an error.
- For a String (see Section 33.2.5.4.9), if the string length is greater than the Maximum Length, then the Forms Processor generates an error.
- For a Number (see Section 33.2.5.4.7), if the number cannot fit in the specified variable storage without loss of significant bits, then the Forms Processor generates an error.
- For all questions, if an `EFI_IFR_INCONSISTENT_IF` evaluates to TRUE, then the Forms Processor will display the specified error text.
- For all questions, if an `EFI_IFR_WARNING_IF` evaluates to TRUE, then the Forms Processor will display the specified warning text.

#### 33.2.5.9.3 Form-Level Validation

Third, it takes place when exiting the form or when the values are submitted. The error occurs under two conditions:

- For all questions, if an `EFI_IFR_NO_SUBMIT_IF` evaluates to TRUE, then the Forms Processor will display the specified error text.
• If a Forms Processor such as a script processor performs Form-Level validation, where the concept of a form is not maintained, then the Form-Level validation must occur before processing question values from other forms or before completion of the configuration session.

33.2.5.10 Forms Processing

Forms Processors interpret the IFR in order to extract information about configuration settings. This section describes how the IFR should be interpreted and how errors should be handled.

33.2.5.10.1 Error Handling

The Forms Processor may encounter problems in interpreting the IFR. This section describes the standard ways of handling these issues:

- **Unknown Opcodes.**
  Unknown opcodes have a type which is not recognized by the Forms Processor. In general, the Forms Processor ignores the opcode, along with any nested opcodes.

- **Malformed Opcodes.**
  Malformed objects have a length which is less than the minimum length for that object type. In this case, the entire form is disabled.

- **Extended Opcodes.**
  Extended objects have a length longer than that expected by the Forms Processor. In this case, the Forms Processor interprets the object normally and ignores the extra data.

- **Malformed Forms Sets**
  Malformed forms sets occur when an object's length would cause it extend beyond the end of the forms set, or when the end of the forms set occurs while a scope is still open. In this case, the entire forms set is ignored.

- **Reserved Bits Set.**
  The Forms Processor should ignore all set reserved bits.

33.2.5.11 Forms Editing

This section describes considerations for Forms Editors, which are a specialized Forms Processor which can create and manipulate form lists, forms and questions in their binary form.

33.2.5.11.1 Locking

Locking indicates that a question or statement—along with its related options, prompts, help text or images—should not be moved or edited. A statement or question is locked when the `IFR_LOCKED` opcode is found within its scope.

UEFI-compliant Forms Editors must allow statements or questions within an image to be locked, but should not allow them to be unlocked. UEFI-compliant Forms Editors must not allow modification of locked statements or questions or any of their associated data (including options, text or images).
Note: This mechanism cannot prevent unauthorized modification. However, it does clearly state the intent of the driver creator that they should not be modified.

33.2.5.11.2 Moving Forms

When forms are moved between form sets, the related data (such as forms, variable stores and default stores) need to have their references renumbered to avoid conflicts with identifiers in the new form set. For forms, these include:

- **EFI_IFR_FORM** or **EFI_IFR_FORM_MAP** (and all references in **EFI_IFR_REF**)
- **EFI_IFR_DEFAULTSTORE** (and all references in **EFI_IFR_DEFAULT**)
- **EFI_IFR_VARSTORE_x** (and all references within question headers)

33.2.5.11.3 Moving Questions

When questions are moved between form sets, the related data (such as images and strings) need to be moved and references to results-processing and storage may need to be revised. For example:

**String and Images.**

If the question is being moved to another form set, then all strings and images associated with the question must be moved to the package list containing the form set and removed from the current one.

**Form Set.**

If the question is moved to a package list installed by a different driver, then the **EFI_IFR_VAR_STORAGE_DEVICE** (see Section 33.3.8.3.92) should be nested in the scope of the question, describing the driver installation device path.

**Question References.**

If a question value in another form set is referred to in any expressions (such as **EFI_IFR_INCONSISTENT_IF** or **EFI_IFR_NO_SUBMIT_IF** or **EFI_IFR_WARNING_IF**) using either **EFI_IFR_QUESTION_REF2** (see Section 33.3.8.3.56) or **EFI_IFR_QUESTION_REF1** (see Section), then these must be converted to a form of **EFI_IFR_QUESTION_REF3** (see Section 33.3.8.3.57), specifying the EFI_GUID of the form set and/or the device path of the package list containing the form set wherein the question referred to is defined.

When questions are moved between forms, whether in the same form list or another form list, question behavior reliant on the current form may need revision. One example is the use of **EFI_IFR_RULE_REF** in expressions. Here, rules are shortcuts for common expressions used in a form. If a question is moved to another form, the references to any rules in expressions must be replaced by the expression itself.

33.2.5.12 Forms Processing & Security Privileges

The IFR provides a way for a Forms Processor to identify which forms, statements, questions and even question values are available only to users with specific privilege levels and enforce those privilege levels.

Setup access security privileges are described in terms of GUIDs. The current user profile either has the specified privilege or it does not. The **EFI_IFR_SECURITY** opcode returns whether or not the current user profile has the specified setup access privilege. Combined with the expressions such as **EFI_IFR_DISABLE_IF**, **EFI_IFR_SUPPRESS_IF**, **EFI_IFR_GRAY_OUT_IF**, **EFI_IFR_WARNING_IF**, etc.
EFI_IFR_INCONSISTENT_IF and EFI_IFR_NOSUBMIT_IF, the author of a form can control access to specific forms, statements and questions, or even control whether specific values are valid.

Forms Processors on systems with multiple setup-related user privilege levels must support report these correctly when processing the EFI_IFR_SECURITY opcode.

Forms Processors on systems which support the UEFI User Authentication proposal must correctly inquire from the current user profile whether or not it has security privileges (see Section 36.4.1.6 on EFI_USER_INFO_ACCESS_SETUP and Section 36.3.1 on EFI_USER_MANAGER_PROTOCOL.GetInfo()).

Forms Processors on systems which support re-identification during the platform configuration process must support reevaluation of the EFI_IFR_SUPPRESS_IF and EFI_IFR_GRAY_OUT_IF upon receipt of notification that the current user profile has been changed by using the UEFI Boot Service CreateEventEx() and the EFI_USER_PROFILE_CHANGED_EVENT_GUID.

33.2.6 Strings

Strings in the UEFI environment are defined using UCS-2, which is a 16-bit-per-character representation. For user-interface purposes, strings are one of the types of resources which can be installed into the HII Database (see Section 33.2.9).

In order to facilitate localization, users reference strings by an identifier unique to the package list which the driver installed. Each identifier may have several translations associated with it, such as English, French, and Traditional Chinese. When displaying a string, the Forms Browser selects the actual text to display based on the current platform language setting.

The actual text for each language is stored separately (in a separate package), which makes it possible to add and remove language support just by including or excluding the appropriate package.
Each string may have font information, including the font family name, font size and font style, associated with it. Not all platforms or displays can support fonts and styles beyond the system default font (see Section 33.2.7), so the font information associated with the string should be viewed as a set of hints.

### 33.2.6.1 Configuration Language Paradigm

This specification uses the RFC 4646 language naming scheme to identify the language that a given string is associated with. Since RFC 4646 allows for the same Primary language tags to contain a large variation of subtags (e.g. regions), a best matching language algorithm is defined in RFC 4647. Callers of interfaces that require RFC 4646 language codes to retrieve a Unicode string, must use the RFC 4647 algorithm to lookup the Unicode string with the closest matching RFC 4646 language code.

Since the majority of strings discussed in this specification are associated with generating a user interface, the languages that are typically associated with strings have commonly defined languages such as en-US, zh-Hant, and it-IT. The RFC 4646 standard also reserves for private use languages prefixed with a value of “x”.

**Note:** This specification defines for its own purposes one of these private use areas as a special-purpose language that components can use for extracting information out of. Assume that any private-use languages encountered by a compliant implementation will likely consider those languages as configuration languages, and the associated behavior when referencing those languages will be platform specific. Section 33.2.11.2 describes an example of such a use.

### 33.2.6.2 Unicode Usage

This section describes how different aspects of the Unicode specification related to the strings within this specification.

#### 33.2.6.2.1 Private Use Area

Unicode defines a private use area of 6500 characters that may be defined for local uses. Suggested uses include Egyptian Hieroglyphics; see *Developing International Software For Windows 95* and *Windows NT* for more information. UEFI prohibits use of this area in a UEFI environment. This is because a centralized font database accumulated from the various drivers (a valid implementation) would end up with collisions in the private use area, and, generally, an XML browser could not display these characters.

#### 33.2.6.2.2 Surrogate Area

The Unicode specification has two 16-bit character representations: UCS-2 and UTF-16. The UEFI specification uses UCS-2. The primary difference is that UTF-16 defines surrogate areas (see page 56 in *Professional XML*) that allow for expanded character representations of the 16-bit Unicode. These character representations are very similar to Double Byte Character Set (DBCS)—2048 Unicode values split into two groups (D800–DBFF and DC00–DFFF). They are defined as having 16 additional bits of value to make up the character, for a total of about one million extra characters. UEFI does not support surrogate characters.

#### 33.2.6.2.3 Non-Spacing Characters

Unicode uses the concept of a nonspacing character. These glyphs are used to add accents, and so on, to other characters by what amounts to logically OR’ing the glyph over the previous glyph. There does not appear to be any predictable range in the Unicode encoding to determine nonspacing characters, yet
these characters appear in many languages. Further, these characters enable spelling of several
languages including many African languages and Vietnamese.

33.2.6.2.4 Common Control Codes

This specification allows the encoding of font display information within the strings using special control
characters. These control codes are meant as display hints, and different platforms may ignore them,
depending on display capabilities.

In single-byte encoding, these are in the form 0x7F 0xyy or 0x7F 0x0y 0xzz. Single-byte encoding is
used only when coupled with the Standard Compression Scheme for Unicode, described in
Section 33.3.6.3.

In double-byte encoding, these are in the form 0xF6yy, 0xF7zz or 0xF8zz. When converted to UCS-2,
all control codes should use the 0xFxyy form.

Table 33-3 Common Control Codes for Font Display Information

<table>
<thead>
<tr>
<th>Value</th>
<th>Description</th>
<th>Single-Byte Encoding</th>
<th>Double-Byte Encoding</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x00</td>
<td>Font Family Select. The subsequent text will be displayed in the font specified by the following byte.</td>
<td>0x7F 0x00 0xzz</td>
<td>0xF7zz</td>
</tr>
<tr>
<td>0x01</td>
<td>Font Size Select. The subsequent text will be displayed in the point size, in half points, specified by the following byte.</td>
<td>0x7F 0x01 0xzz</td>
<td>0xF8zz</td>
</tr>
<tr>
<td>0x20</td>
<td>Bold On.</td>
<td>0x7F 0x20</td>
<td>0xF620</td>
</tr>
<tr>
<td>0x21</td>
<td>Bold Off</td>
<td>0x7F 0x21</td>
<td>0xF621</td>
</tr>
<tr>
<td>0x22</td>
<td>Italic On</td>
<td>0x7F 0x22</td>
<td>0xF622</td>
</tr>
<tr>
<td>0x23</td>
<td>Italic Off</td>
<td>0x7F 0x23</td>
<td>0xF623</td>
</tr>
<tr>
<td>0x24</td>
<td>Underline On</td>
<td>0x7F 0x24</td>
<td>0xF624</td>
</tr>
<tr>
<td>0x25</td>
<td>Underline Off</td>
<td>0x7F 0x25</td>
<td>0xF625</td>
</tr>
<tr>
<td>0x26</td>
<td>Emboss ON</td>
<td>0x7F 0x26</td>
<td>0xF626</td>
</tr>
<tr>
<td>0x27</td>
<td>Emboss OFF</td>
<td>0x7F 0x27</td>
<td>0xF627</td>
</tr>
<tr>
<td>0x28</td>
<td>Shadow ON</td>
<td>0x7F 0x28</td>
<td>0xF628</td>
</tr>
<tr>
<td>0x29</td>
<td>Shadow OFF</td>
<td>0x7F 0x29</td>
<td>0xF629</td>
</tr>
<tr>
<td>0x2A</td>
<td>DblUnderline ON</td>
<td>0x7F 0x2A</td>
<td>0xF62A</td>
</tr>
<tr>
<td>0x2B</td>
<td>DblUnderline OFF</td>
<td>0x7F 0x2B</td>
<td>0xF62B</td>
</tr>
</tbody>
</table>

33.2.6.2.5 Line Breaks

This section describes the use of control characters to determine where break opportunities within
strings. These guidelines are based on Unicode Technical Report #14, but are significantly simplified.

Spaces

In general, any of the following space characters is a line-break opportunity:

<table>
<thead>
<tr>
<th>Value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>0020</td>
<td>SPACE</td>
</tr>
<tr>
<td>1680</td>
<td>OGHAM SPACE MARK</td>
</tr>
</tbody>
</table>
When a space is desired without a line-break opportunity, one of the following spaces should be used:

<table>
<thead>
<tr>
<th>Character Code</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>00A0</td>
<td>NO-BREAK SPACE (NBSP)</td>
</tr>
<tr>
<td>202F</td>
<td>NARROW NO-BREAK SPACE (NNBSP)</td>
</tr>
</tbody>
</table>

**In-Word Break Opportunities**

In some cases, allowing line-breaks in a word is desirable. These line break opportunities should be explicitly described using one of the characters from the following list:

<table>
<thead>
<tr>
<th>Character Code</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>200B</td>
<td>ZERO WIDTH SPACE (ZWSP)</td>
</tr>
</tbody>
</table>

**Hyphens**

The following characters are hyphens and other characters which describe line break opportunities after the character.

<table>
<thead>
<tr>
<th>Character Code</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>058A</td>
<td>ARMENIAN HYPHEN</td>
</tr>
<tr>
<td>2010</td>
<td>HYPHEN</td>
</tr>
<tr>
<td>2012</td>
<td>FIGURE DASH</td>
</tr>
<tr>
<td>2013</td>
<td>EN DASH</td>
</tr>
<tr>
<td>0F08</td>
<td>TIBETAN MARK INTERSYLLABIC TSHEG</td>
</tr>
<tr>
<td>1361</td>
<td>ETHIOPIC WORDSPACE</td>
</tr>
<tr>
<td>17D5</td>
<td>KHMER SIGN BARIYOOSAN</td>
</tr>
</tbody>
</table>

The following characters describe line break opportunities before and after them, but not between a pair of them:

<table>
<thead>
<tr>
<th>Character Code</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>2014</td>
<td>EM DASH</td>
</tr>
</tbody>
</table>
The following characters describe a hyphen which is not a line-breaking opportunity:

| 2011 | NON-BREAKING HYPHEN (NBHY) |

Mandatory Breaks
The following characters force a line-break:

| 000A | NEW LINE |
| 000C | FORM FEED |
| 000D | CARRIAGE RETURN |
| 2028 | LINE SEPARATOR |
| 2029 | PARAGRAPH SEPARATOR |

33.2.7 Fonts
This section describes how fonts are used within the UEFI environment.

UEFI describes a standard font, which is required for all systems which support text display on bitmapped output devices. The standard font (named ‘system’) is a fixed pitch font, where all characters are either narrow (8x19) or wide (16x19). UEFI also allows for display of other fonts, both fixed-pitch and variable-pitch. Platform support for these fonts is optional.

UEFI fonts are described using either the Simplified Font Package (Section 33.3.2) or the normal Font Package (Section 33.3.3).

33.2.7.1 Font Attributes
Fonts have the following attributes:

**Font Name**
The font name describes, in broad terms, the visual style of the font. For example, “Arial” or “Times New Roman” The standard font always has the name “sysdefault”.

**Font Size**
The font size describes the maximum height of the character cell, in pixels. The standard font always has the font size of 19.

**Font Style**
The font style describes standard visual modifies to the base visual style of a font. Supported font styles include: bold, italic, underline, double-underline, embossed, outline and shadowed. Some font styles may also be simulated by the font rendering engine. The standard font always has no additional font styles.

33.2.7.2 Limiting Glyphs
Strings in the UEFI environment can be presented in environments with very different limitations. The most constrained environment is in the firmware phases prior to discovery of a boot device with a system partition. The main limitation in this environment is storage space. If unexpected strings could be
displayed before system partition availability, the UEFI environment would have to store glyphs for all characters in a Unicode font. After system partition discovery, all glyphs could be made available.

Careful user interface design can limit to a manageable number, the quantity of unexpected characters that the system could be called on to display. Knowing what strings the firmware is going to display limits the number of glyphs it is required to carry.

In addition, carefully designed firmware can support a system where a limited number of strings are displayed before system partition availability. This may be done while enabling the input and display of large numbers of characters/glyphs using a full font file stored on the system partition. In such a situation, the designer must ensure that enough information can be displayed. The designer must also insure that the configuration can be changed using only information from firmware-based non-volatile storage to obtain access to a satisfactory system partition.

UEFI requires platform support of a font containing the basic Latin character set.

While the system firmware will carry this standard font, there might be times when a UEFI application or driver requires the printing of a character not contained within the platform firmware. In this case, a UEFI driver or application can carry this font data and add it to the font already present in the HII Database. New font glyphs are accepted when there is no font glyph definition for the Unicode character already in the specified font.

In addition the standard system font and fonts extended by UEFI applications or drivers, it is possible for drivers that implement the EFI HII Font Glyph Generator Protocol to render additional font glyphs with specific font name, style, and size information, and add the new font packages to the HII Database. That is when HII Font Ex searches the glyph block in the existing HII font packages, it will try to locate `EFI_HII_FONT_GLYPH_GENERATOR_PROTOCOL` protocol for generating the corresponding glyph block and inserting the new glyph block into HII font package if the glyph block information is not exist in any HII font package. The HII font package which the new glyph block inserted can be an existing HII font package or a new HII font package created by HII Font Ex according to the `EFI_FONT_DISPLAY_INFO` of character.

The figure below shows how fonts interact with the HII database and UEFI drivers, even if the font does not already exist in the database.
33.2.7.3 Fixed Font Description

To allow a UEFI application or driver to extend the existing fonts with additional characters, the UEFI driver must be able to provide characters that fit aesthetically with the system font. For this reason the capability to define attributes of different fonts and to suggest a reasonable default target for these parameters is important.

Fonts can vary in width, style, baseline, height, size, and so on. The fixed font definition includes white space and the glyph data, as well as the positioning of the glyph data. This prevents characters of different fixed fonts from being adjusted at runtime to fit aesthetically together. To provide UEFI drivers
with a basic description of how to design fixed font characters, a subset of industry standard font terms are defined below:

**baseline**

The distance from upper left corner of cell to the base of the Caps (A, B, C, …)

**cap_height**

The distance from the base of the Caps to the top of the Caps

**x_height**

The distance from the baseline to the top of the lower case ‘x’

**descender**

The distance some characters extended below the baseline (g, j, p, q, y)

**ascender**

The distance from the top of the lower case ‘x’ to the tall lower case characters (b, d, f, h, k, l)

The following figure illustrates the font description terms:

![Figure 33-19 Font Description Terms](image)

This 8x19 system font example (above), follows the original VGA 8x16 definition and creating double wide vertical lines, giving a **bold** look to the font (style = bold). Along with matching the 8x19 base system font, if a UEFI driver wants to extend the DBCS (Double Byte Character Set) font, it must be aware of the parameters that describe the 16x19 font, as shown below.
This 16x19 font example (above) has a style of *plain* (single width vertical lines) instead of *bold* like the 8x19 font, since there is not enough horizontal resolution to cleanly define the DBCS glyphs. The 16x19 ASCII characters have also been designed in a style matching the DBCS characters, allowing them to fit aesthetically together. Note that the default 16x19 fixed width characters are not stored like 1-bit images, one row after another; but instead stored with the left column (19 bytes) first, followed by the right column (19 bytes) of character data. The figure below shows how the characters of the previous figure would be laid out in the font structure.
33.2.7.3.1 System Fixed Font Design Guidelines

To allow a UEFI application or driver to extend the fixed font character set, the UEFI system fonts must adhere, at least roughly, to the design guidelines in the table below:

<table>
<thead>
<tr>
<th>Term</th>
<th>8 x 19 Font</th>
<th>16 x 19 Font</th>
</tr>
</thead>
<tbody>
<tr>
<td>baseline</td>
<td>15 pixels</td>
<td>14 pixels</td>
</tr>
<tr>
<td>cap_height</td>
<td>12 pixels</td>
<td>11 pixels</td>
</tr>
<tr>
<td>x_height</td>
<td>8 pixels</td>
<td>7 pixels</td>
</tr>
<tr>
<td>descender</td>
<td>3 pixels</td>
<td>4 pixels</td>
</tr>
<tr>
<td>ascender</td>
<td>4 pixels</td>
<td>4 pixels</td>
</tr>
</tbody>
</table>

In the table above lists the terms in priority order. The most critical guideline to match is the baseline, followed by cap_height and x_height. The terms descender and ascender are not as critical to the aesthetic look of the font as are the other terms. These font design parameters are only guidelines. Failing to match them will not prevent reasonable operation of a UEFI driver that attempting to extend the system font.

33.2.7.4 Proportional Fonts Description

Unlike the fixed fonts, proportional fonts do not have a predefined character cell; instead the character cell is created based on the characters that are being displayed in the current line. In a proportional font only the glyph data is defined, no whitespace. Instead, the proportional font defines five parameters (Width, Height, Offset_X, Offset_Y, & Advance), which allow the glyph data to be position in the character cell and calculate the origin of the next character.

In the figure below, you can see these parameters (in ‘…’) for the characters shown, in addition you can see the actual byte storage (the padding to the nearest byte is shown shaded).

To determine font baseline, scan all font glyphs calculating sum of Height and Offset_Y for each glyph. The largest value of the sum defines location of the baseline.

The font line height is calculated by adding baseline with the largest by absolute value negative Offset_Y among all the font glyphs.
33.2.7.4.1 Aligning Glyphs to the Baseline

To display a line of proportional glyphs, baseline and line height have to be determined. If all the characters to be displayed are from the same font, the baseline and line height are the baseline and line height of the font.

If the characters being displayed are from different fonts, scan glyphs of the characters to be displayed calculating sum of Height and Offset_Y for each glyph. The largest value of the sum defines location of the baseline.

The line height is calculated by adding baseline with the largest by absolute value negative Offset_Y among all the characters to be displayed.

As shown in the following figure, once the baseline value is found it is added to the starting position of the line to calculate the Origin. From the Origin, each and every glyph can be generated based on the individual glyph parameters, including the calculation of the next glyph’s Origin.

![Figure 33-23 Aligning Glyphs](image)

The starting position (upper left hand corner) of the glyph is defined by (Origin_X + Offset_X), (Origin_Y – (Offset_Y + Height)). The Origin of the next glyph is defined by (Origin_X + Advance), (Origin_Y).

In addition to determining the line height and baseline values; the scan of the characters also calculates the line width by totaling up all of the advance values.

33.2.7.4.2 Proportional Font Design Guidelines

This method of aligning glyphs to a baseline allows one to place wildly different characters correctly position on a single line. However there still is a need for the system proportional fonts to roughly adhere to overall font height (19 pixels high character cells) and the placement of the baseline at the bottom of the Caps (if applicable or about 5 pixels up from the bottom of the character cell). These guidelines are not as critical as the fixed font guidelines, since the character cell height are defined at runtime, based on what else is displayed with that character.

33.2.8 Images

The format of the images to be stored in the Human Interface Infrastructure (HII) database have been created to conform to the industry standard 1-bit, 4-bit, 8-bit, and 24-bit video memory layouts. The 24-bit and 32-bit display systems have the exact same display capabilities and the exact same pixel definition. The difference is that the 32-bit pixels are DWORD aligned for improve CPU efficiency when accessing video memory. The extra byte that is inserted from the 24-bit and the 32-bit layout has no bearing on the actual screen.
Video memory is arranged left-to-right, and then top-to-bottom. In a 1-bit or monochrome display, the most significant bit of the first byte defines the screen’s upper left most pixel. In a 4-bit or 16 color display, the most significant nibble of the first byte defines the screen’s upper left most pixel. In an 8-bit or 256 color display, the first byte defines the screen’s upper left most pixel.

In both the 24-bit and 32-bit TrueColor displays, the first three bytes define the screen’s upper left most pixel. The first byte is the pixel’s blue component value, the next byte is the pixel’s green component value, and the third byte is the pixel’s red component value (B,G,R). Each color component value can vary from 0x00 (color off) to 0xFF (color full on), allowing 16.8 millions colors that can be specified. In the 32-bit TrueColor display modes, the fourth byte is a don’t care.

33.2.8.1 Converting to a 32-bit Display

The UEFI recommended video mode for computer-like devices uses a 32-bit Linear Frame Buffer video mode. All images stored in the HII database will need conversion to 32-bit before display.

To display a 24-bit image into 32-bit video memory, a pixel of the image is retrieved (read DWORD value advance pixel offset by 3) and then written to the video memory (write DWORD value advance pixel offset by 4).

To display any of the non-TrueColor images (1-bit, 4-bit, and 8-bit), there is an extra step of indirection through the palette definition to get the TrueColor pixel value. First retrieve the palette index value by isolating the corresponding bits, then index into the associated palette to retrieve the 24-bit (B,G,R) color entry (read DWORD value), then write it to the video memory (write DWORD value advance pixel offset by 4). For this reason, the palette color entry definition is defined exactly the same as the image color pixel (B,G,R).

33.2.8.2 Non-TrueColor Displays

It is possible to display the HII database images on non-TrueColor video modes. You cannot however, display images beyond the bit depth of the target screen resolution. For example you would be able to display 1-bit, 4-bit, and 8-bit images in a 256 color video mode. To do this you must create a global palette (256 entries), by merging all images color needs to a best fit palette and then programming the hardware palette with that data.

The hardware palette color definition (R,G,B) is backwards from the screen pixel definition (B,G,R), and will have to be swapped before programming. In addition, the hardware palette may only support 6-bit of magnitude per color component instead of the 8-bit defined in the palette information section; therefore the values will have to be shifted before writing.

33.2.9 HII Database

The Human Interface Infrastructure (HII) database is the resource that serves as the repository of all the form, string, image and font data for the system. Drivers that contain information that is appropriate for the database will export this data to the HII database.

For example, one driver might contain all the motherboard-specific data (the traditional “Setup” for the system). Additionally, add-in cards may contain their own drivers, which, in turn, have their own Setup-related data. All of the drivers that contain Setup-related data would export their information to the HII database, as shown in the figure below.
33.2.10 Forms Browser

The UEFI Forms Browser is the service that reads the contents of the HII Database and interprets the forms data in order to present it to the user. For example, the Forms Browser can be used to gather all setup-related data and presents it to the user. This service also takes the user input and allows for changes to be saved into non-volatile storage.

The figure below shows the relationship between the HII database, UEFI drivers, and the UEFI Forms Browser.

33.2.10.1 User Interaction

The Forms Browser implementer has great flexibility as to the type of actual user interface provided. For example, while required to support some forms of navigation (see `EFI_FORM_BROWSER2_PROTOCOL.SendForm()` or the cross-reference question), it may optionally support additional navigation capabilities, such as a back button or a menu bar. This section describes the rules to which the Forms Browser user-interaction must conform.
33.2.10.1.1 Forms Browser details

The forms browser maintains a collection of one or more forms. The forms browser is required to provide navigation for these forms if there is more than one (see Section 35.6, “Form Browser Protocol”).

The forms browser maintains one or more active forms. An active form is any form where the forms browser is maintaining a set of question values. A form is considered active after all question values have been read from storage and the EFI_BROWSER_ACTION_FORM_OPEN action has been sent to all questions on the form which require callback. A form is considered inactive after all question values have been either discarded or written to storage and the EFI_BROWSER_ACTION_FORM_CLOSE action has been sent to all questions on the form which require callback.

The forms browser maintains a selected form. The selected form contains the selected question and indicates the primary area of user interaction.

The standards form navigation behaviors are:

Navigate Forms.

When the user chooses this required behavior, a new form is selected and, if any questions on the form are selectable (see Section 33.2.5.3.4), a question is selected. Forms browsers are required to provide navigation to (at least) the first form in all form sets when FormId is zero (see Section 35.6). This behavior cannot be selected if the current form is modal (see Section 33.2.5.2, “Forms”).

Exit Browser/Discard All.

When the user chooses this optional behavior, the question values for active forms are discarded, the active forms are deactivated and the forms browser exits with an action request of EFI_BROWSER_ACTION_REQUEST_EXIT. This behavior cannot be selected if the current form is modal (see Section 33.2.5.2, “Forms”).

Exit Browser/Submit All.

When the user chooses optional behavior, the question values are written to storage, the active forms are deactivated and the forms browser exits with an action request of EFI_BROWSER_ACTION_REQUEST_SUBMIT or EFI_BROWSER_ACTION_REQUEST_RESET. This behavior cannot be selected if the current form is modal (see Section 33.2.5.2, “Forms”).

Default.

When the user chooses this optional behavior, the current question values for the questions on the focus form are updated from one of the default stores and then the EFI_IFR_BROWSER_ACTION_REQUEST_DEFAULT_x action is sent for each of the questions with the Callback attribute. This behavior can be initiated by a Reset Button question (see section Section 33.2.5.3.8).

33.2.10.1.2 Selected Form

When a form is made active, the forms browser sends the EFI_BROWSER_ACTION_FORM_OPEN for all questions supporting callback, retrieves the current question values, saves those as the original question values and begins refreshing any questions that support it.
The forms browser maintains a current question value for each question on active forms. The current question value is the last value that the forms browser read from storage/callback (see Section 33.2.5.4.1, “Values”) or the last value committed by the user. The form is considered modified if any of the current question values are modified (see Questions, below). The forms browser refreshes the current question values of at least questions on the selected with a non-zero refresh interval.

The forms browser maintains a selected question on the selected form. The selected question is the primary focus of the user’s interaction. When a form is selected, the forms browser must choose a selectable question (see Section 33.2.5.3.4, “Evaluation of Selectable Statements”) as the selected question, if one is present on the form.

The standard active form behaviors are:

**Exit Browser/Discard All.**
When the user chooses this required behavior, the question values for active forms are discarded, the active forms are deactivated and the forms browser exits with an action request of `EFI_BROWSER_ACTION_REQUEST_EXIT`. This behavior can be initiated by the function associated with a question with the Callback attribute.

**Exit Browser/Submit All.**
When the user chooses this required behavior, the current question values for active forms are validated (see nosubmitif, Section 33.3.8.3.45) and, if successful, question values for active forms are written to storage, the active forms are deactivated and the forms browser exits with an action request of `EFI_BROWSER_ACTION_REQUEST_SUBMIT`. This behavior can be initiated by the function associated with a question with the Callback attribute.

**Exit Browser/Discard All/Reset Platform.**
When the user chooses this required behavior, the question values for active forms are discarded, the active forms are deactivated and the form browser exits with an action request of `EFI_BROWSER_ACTION_REQUEST_RESET`. This behavior can be initiated by the function associated with a question with the Callback attribute.

**Exit Form/Submit Form.**
Apply Form. When the user chooses this required behavior, the question values for the selected form are validated (see -nosubmitif, BUGBUG< ) and, if successful, question values for the selected form are written to storage and the selected form is deselected. This behavior can be initiated by the function associated with a question with the Callback attribute.

**Exit Form/Discard Form.**
When the user chooses this required behavior, the question values for the selected form are discarded and the selected form is deselected. This behavior can be initiated by the function associated with a question with the Callback attribute.

**Apply Form.**
When the user chooses this required behavior, the question values for the selected form are validated (see nosubmitif, BUGBUG) and, if successful, question values for
the selected form are written to storage. This behavior can be initiated by the function associated with a question with the Callback attribute.

**Discard Form.**

When the user chooses this required behavior, the question values for the selected form are discarded. This behavior can be initiated by the function associated with a question with the Callback attribute.

**Default.**

When the user chooses this required behavior, the current question values for the questions on the selected form are updated from a default store. This behavior can be initiated by a Reset Button question (see Section 33.2.5.3.8).

**Navigate To Question.**

When the user chooses this required behavior, the selected question is deselected and another question on the same form is selected. The types of navigation provided between questions on the same form are beyond the scope of this specification.

**Navigate To Form.**

When the user chooses this required behavior, the selected form is deselected and the form specified by the question is selected. This behavior can be initiated by a Cross-Reference question. Note that this behavior is distinct from the Navigate Forms behavior described in Forms Navigation.

From these basic behaviors, more complex behaviors can be constructed. For example, a forms browser might check whether the form is modified and, if so, prompt the user to select between the Exit Browser/Discard All and Exit Browser/Submit All behaviors.

### 33.2.10.1.3 Selected Question

When the user navigates to a question or the forms browser selects a form with a selectable question, the forms browser places the question in the **static** state. When the user is choosing another question values for the selected question (by typing or from a menu or other means), the forms browser places the question in the **changing** state. When the user finalizes selection of a question value the forms browser returns the question to the static state.

The forms browser refreshes all questions in at least the selected form with a non-zero refresh interval that are not modified. Typically, a forms browser will not update the displayed question value while the selected question is in the changing state, but will when the selected question is in the static state. A question is considered modified if there is storage associated with the question (i.e., a variable store was specified) and the current question value is different from the original question value.

The standard active question behaviors are:

**Change**

When the user chooses this required behavior, the forms browser places the selected question in the changing state and allows the user to specify a new current question value for the active question. For example, selecting items in a drop box or beginning to type a new value in an edit box.
With some question types and user interface styles, this behavior is hidden from the user. For example, with check boxes or radio buttons as found in most windowed user-interfaces, the user changes and commits the value with one action. Likewise, with action buttons, selecting the action button implies both the question value and the commit action.

This behavior corresponds to the CHANGING browser action request for questions that support callback.

**Commit**

When the user chooses this required behavior, the forms browser validates the specified question value (see `EFI_IPF_INCONSISTENT_IF`, Section 33.3.8.3.33) and, if successful, places the selected question in the static state and updates the current question value to that specified while in the changing state. If the selected question’s current question value is different than the selected question’s original question value, the selected question is considered modified. The form browser must then re-evaluate the modifiability, selectability and visibility of other questions in the selected form.

This behavior corresponds to the CHANGED browser action request for questions that support callback.

**Discard**

When the user chooses this required behavior, the forms browser places the question in the changed state.

### 33.2.11 Configuration Settings

In order to save user changes to configuration settings after the system reset or power-off, there must be some form of non-volatile storage available. There are two types of non-volatile storage: system non-volatile storage or add-in card non-volatile storage. Both types are supported.

In general, settings are not saved to non-volatile storage until the user specifically directs the Forms Browser to do so. There are exceptions, such as when operating in a batch or script mode, setting a system password, and updating the system date and time. The underlying platform support dictates whether or not hardware configuration changes are committed immediately.

As shown in the figure below, when a system reset occurs, the firmware’s initialization routines will launch the UEFI drivers (e.g. option ROMs). Drivers enabled to take direction from a non-volatile setting read the updated settings during their initialization.
33.2.11.1 OS Runtime Utilization

Due to the static nature of the data that is contained in the HII Database and the fact that certain classes of non-volatile storage can be updated during OS run-time, it is possible for an application running under an OS to read the HII information, make configuration changes and even make changes.

The figure below shows how an OS makes use of the HII database during runtime. In this case, the contents of the HII Database is exported to a buffer. The pointer to the buffer is placed in the EFI System Configuration Table, where it can be retrieved by an OS application.
The process used to allow an OS application to use this is as follows:

Drivers/applications in the system register user interface data into the HII Database

When the platform transitions from pre-boot to runtime phases of operation, the HII `ExportPackageLists()` is called to export the contents of the HII Database into a runtime buffer.

This runtime buffer is advertised in the UEFI Configuration Table using the HII Database Protocol’s GUID so that an OS application can find the data.

The HII `ExportConfig()` is called to export the current configuration into a runtime buffer.

This runtime buffer is advertised in the UEFI Configuration Table using the HII Configuration Routing Protocol’s GUID so that an OS application can find the data.

When an O/S application wants to display pre-boot configuration content, it searches the UEFI Configuration Table for the HII Database Protocol’s GUID entry and renders the contents from the runtime buffer which it points to.

If the OS application needs to update the system configuration, the configuration information can be updated.

For those configuration settings which are stored in UEFI variables (i.e., using `GetVariable()` and `SetVariable()`), the application can update these using the abstraction provided by the operating system.

For those configuration settings which are not stored in UEFI variables, the OS application can use the UEFI UpdateCapsule runtime service to change the configuration.
33.2.11.2 Working with a UEFI Configuration Language

By defining the concept of a language that may provide hints to a consumer that the string payload may contain pre-defined standard keyword content, the user of this solution can export their configuration data for evaluation. This evaluation enables the consumer to determine if a particular platform supports a given configuration language, and in-turn be able to adjust known settings that are stored in a platform-specific manner. An example of this is illustrated below which uses various component described in this and the other HII chapters of this specification. In the example, a fictional technology called XYZ exists, and this particular platform supports it. The question is, how does a standard application which is not privy to the platform’s construction know how this setting is stored? To-date, this is not a reasonably solvable problem, but in the illustration below, this example shows how one might go about solving this issue.

Figure 33-28 Standard Application Obtaining Setting Example

33.2.12 Form Callback Logic

Since it has been the design intent that the forms processor not need to understand the underlying hardware implementations or design paradigms of the platform, there were certain needs that could
only be met by calling a more platform knowledgeable component. In this case, the component would typically be associated with some hardware device (e.g. motherboard, add-in card, etc.). To facilitate this interaction, some formal interfaces were declared for more platform-specific components to advertise and the forms processor could then call.

Note that the need for the forms processor to call into an alternate component driver should be limited as much as possible. The two primary reasons for this are the cases where off-line or O/S-present configuration is important. The three flow charts which follow describe the typical decisions that a forms processor would make with regards to handling processes which necessitate a callback.

Figure 33-29 Typical Forms Processor Decisions Necessitating a Callback (1)
Figure 33-30 Typical Forms Processor Decisions Necessitating a Callback (2)
33.2.13 Driver Model Interaction

The ability for a UEFI driver to interact with a target controller is abstracted through the Configuration Access Protocol. If a particular piece of hardware managed by a controller needs configuration services, it is the responsibility of that controller to provide this configuration abstraction for the given device. Regardless of whether a device driver or bus driver is abstracting the hardware configuration, the interaction with a configured device is identical.

Note that the ability for a driver to provide these access protocols might be done fairly early in the initialization process. Depending on the hardware capabilities, one might be advantaged in providing configuration access very early so that being able to determine a given device’s current settings can be done without a full enumeration of certain bus devices. Also note that the same recommendations that are made in the DriverBinding sections should still be maintained. These cover the Supported, Started, and Stopped functions.
33.2.14 Human Interface Component Interactions

The figure below depicts the model used inside a common deployment of HII to manage human interface components.
33.2.15 Standards Map Forms

Configuration settings are configuration settings. But the way in which they are controlled is driven by different requirements. For example, the UEFI HII infrastructure focuses primarily on the way in which the configuration settings can be browsed and manipulated by a user. Other standards such as the DMTF Command-Line Protocol, focus on the way in which configuration settings can be manipulated via text commands.

Each configuration method tends to view the configuration settings a different way. In the end, they are changing the same configuration setting, but their means of exposing the control differs. The means by which a configuration method (HII, DMTF, WMI, SNMP, etc.) exposes an individual configuration setting is called a question.

In many cases, there is a one-to-one mapping between the questions exposed by these different configuration methods. That is, a question, as exposed by one configuration method matches the semantic meaning of the configuration setting exactly.

However, in other cases, there is not a one-to-one mapping. These cases break down into three broad categories:

1. Value Shift. In this case, the configuration setting has the same scope as the question exposed by a configuration method, but the values used to describe them are different. It may be as simple as 1=5, 2=6, 3=7, etc. or something more complicated, where “ON”=1 and “OFF”=0.

2. One-To-Many. In this case, the configuration setting maps to two or more questions exposed by a configuration method. For example the configuration setting might have the following enumerated values:
   a  0 = Disable Serial Port
   b  1 = Enable Serial Port, I/O Port 0x3F8, IRQ 4
c 2 = Enable Serial Port, I/O Port 0x2F8, IRQ 3  

d 3 = Enable Serial Port, I/O Port 0x3E8, IRQ 4  
e 4 = Enable Serial Port, I/O Port 0x2E8, IRQ 3

But in the configuration method, the serial port is controlled by three separate questions:

- Question #1: 0 = disable, 1 = enable  
- Question #2: I/O Port (disabled if Question #1 = 0)  
- Question #3: IRQ (disabled if Question #1 = 0)

Changing the configuration method question #1 to a value of 0 requires that the configuration setting be set to 0. In this case, there is the possibly of data loss. After changing the configuration setting to 0, the information about the I/O port and IRQ are not preserved.

So, in order to change the configuration setting to the value of 1 would require three of the configuration method’s questions to change value: Question #1=1, Question #2=0x3F8, Question #3=IRQ 4.

Figure 33-34 EFI IFR Form set configuration

3. Many-To-One. In this case, the conditions are reversed from the example described in #2 above. Now there are three configuration settings which map to a single configuration method question.

For example, the configuration settings are described using three separate questions:

- Question #1: 0 = disable, 1 = enable  
- Question #2: I/O Port (disabled if Question #1 = 0)  
- Question #3: IRQ (disabled if Question #1 = 0)
But in the configuration method, the serial port is controlled by a single question with the following enumerated values:

a 0 = Disable Serial Port
b 1 = Enable Serial Port, I/O Port 0x3F8, IRQ 4
c 2 = Enable Serial Port, I/O Port 0x2F8, IRQ 3
d 3 = Enable Serial Port, I/O Port 0x3E8, IRQ 4
e 4 = Enable Serial Port, I/O Port 0x2E8, IRQ 3

So, in order to change the configuration method to the value of 1 would require three configuration settings to change value: Question #1=1, Question #2=0x3F8, Question #3=IRQ 4.

Some configuration settings may involve more than one of these mappings.

Standards map forms describe the questions exposed by these other configuration methods and how they map back to the configuration settings exposed by the UEFI drivers. Each standards map form describes the mapping for a single configuration method, along with that configuration method’s name and version.

The questions within standards map forms are encoded using IFR in the same fashion as those within other UEFI forms. The prompt strings for these questions are tied back to the names for those questions within the configuration method (e.g., DMTF CLP).
33.2.15.1 Create A Question’s Value By Combing Multiple Configuration Settings

Rather than reading directly from storage, these standards map questions retrieve their value using the EFI_IFR_READ (Section 33.3.8.3.58) operator. This operator can aggregate a value from more than one configuration settings using EFI_IFR_GET (Section 33.3.8.3.27). This operator can also change the type (integer, string, Boolean) of the value so that, say, a configuration setting with a type of integer can be represented in a standards map form as a string.

For example, to map a single question to three configuration settings (CS1, CS2 and CS3) as described in scenario #3 in Section 33.2.1.5 above would have the following truth table:

<table>
<thead>
<tr>
<th>CS1</th>
<th>CS2</th>
<th>CS3</th>
<th>Q</th>
</tr>
</thead>
<tbody>
<tr>
<td>false</td>
<td>X</td>
<td>X</td>
<td>0</td>
</tr>
<tr>
<td>true</td>
<td>0x3F8</td>
<td>4</td>
<td>1</td>
</tr>
<tr>
<td>true</td>
<td>0x2F8</td>
<td>3</td>
<td>2</td>
</tr>
<tr>
<td>true</td>
<td>0x3E8</td>
<td>4</td>
<td>3</td>
</tr>
<tr>
<td>true</td>
<td>0x2E8</td>
<td>3</td>
<td>4</td>
</tr>
<tr>
<td>true</td>
<td>any other value</td>
<td>any other value</td>
<td>Undefined</td>
</tr>
</tbody>
</table>

These become the following equations:

\[ x_0: \text{Get}(\text{CS1}) ? x_1 : 0 \]
\[ x_1: ((\text{Get}(\text{CS2}) & 0xF00) >> 8) == \text{Get}(\text{CS3}) + 1 ? x_2 : \text{Undefined} \]
\[ x_2: \text{Map}(\text{Get}(\text{CS2}), 0x3f8, 1, 0x2F8, 2, 0x3E8, 3, 0x2E8, 4) \]

33.2.15.2 Changing Multiple Configuration Settings From One Question’s Value

Rather than writing directly to storage, these standards map questions change their value using the EFI_IFR_WRITE (Section 33.3.8.3.94) operator. This operator can, in turn, use the EFI_IFR_SET (Section 33.3.8.3.66) operator to change one or more configuration settings. This operator can also change the type (integer, string, Boolean, etc.) of the value written so that, say, a configuration setting with a type of integer can be represented in a standards map form as a string.

For example, in example #2 above, the following table applies:

<table>
<thead>
<tr>
<th>CS1</th>
<th>CS2</th>
<th>CS3</th>
<th>Q</th>
</tr>
</thead>
<tbody>
<tr>
<td>false</td>
<td>X</td>
<td>X</td>
<td>0</td>
</tr>
<tr>
<td>true</td>
<td>0x3F8</td>
<td>4</td>
<td>1</td>
</tr>
<tr>
<td>true</td>
<td>0x3E8</td>
<td>3</td>
<td>2</td>
</tr>
<tr>
<td>true</td>
<td>0x2F8</td>
<td>4</td>
<td>3</td>
</tr>
<tr>
<td>true</td>
<td>0x2E8</td>
<td>3</td>
<td>4</td>
</tr>
</tbody>
</table>
\textbf{Set}(CS1,Q \neq 0) \&\&
\textbf{Set}(CS2, \text{Map}(\text{this},1,0x3F8,2,0x3E8,3,0x2F8,4,0x2E8)) \&\&
\textbf{Set}(CS3, \text{Map}(\text{this},1,4,2,3,3,4,4,3)

\textbf{33.2.15.3 Value Shifting}

Value shifting is facilitated by the \texttt{EFI_IFR_MAP} (Section 33.3.8.3.38) operator. If this operator finds a value in a list, it replaces it with another value from the list, even if the other value is a different type.

For example, consider the following list of values

\textbf{Table 33-7 Values:}

<p>| | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>PEI Module</td>
</tr>
<tr>
<td>2</td>
<td>DXE Boot Service Driver</td>
</tr>
<tr>
<td>3</td>
<td>DXE Runtime Driver</td>
</tr>
<tr>
<td>10</td>
<td>UEFI Boot Service Driver</td>
</tr>
<tr>
<td>11</td>
<td>UEFI Runtime Driver</td>
</tr>
<tr>
<td>12</td>
<td>UEFI Application</td>
</tr>
</tbody>
</table>

If the integer value 10 were supplied, the value “UEFI Boot Service Driver” would be returned. If the integer value 20 were supplied, Undefined would be returned.

\textbf{33.2.15.4 Prompts}

In standards map forms, the prompts can be used as the key words for the configuration method. They should be specified in the language \texttt{i-uefi} unless there are multiple translations available. Other standards may use the question identifiers as the means of identifying the standard question.

\textbf{33.3 Code Definitions}

This chapter describes the binary encoding of the different package types:

- Font Package
- Simplified Font Package
- String Package
- Image Package
- Device Path Package
- Keyboard Layout Package
- GUID Package
- Forms Package
33.3.1 Package Lists and Package Headers

**EFI_HII_PACKAGE_HEADER**

**Summary**
The header found at the start of each package.

**Prototype**
```c
typedef struct {
    UINT32 Length:24;
    UINT32 Type:8;
    UINT8 Data[ ... ];
} EFI_HII_PACKAGE_HEADER;
```

**Members**
- **Length**
  The size of the package in bytes.
- **Type**
  The package type. See `EFI_HII_PACKAGE_TYPE_x`, below.
- **Data**
  The package data, the format of which is determined by `Type`.

**Description**
Each package starts with a header, as defined above, which indicates the size and type of the package. When added to a pointer pointing to the start of the header, `Length` points at the next package. The package lists form a package list when concatenated together and terminated with an `EFI_HII_PACKAGE_HEADER` with a `Type` of `EFI_HII_PACKAGE_END`.

The type `EFI_HII_PACKAGE_TYPE_GUID` is used for vendor-defined HII packages, whose contents are determined by the `Guid`.

The range of package types starting with `EFI_HII_PACKAGE_TYPE_SYSTEM_BEGIN` through `EFI_HII_PACKAGE_TYPE_SYSTEM_END` are reserved for system firmware implementers.
Related Definitions

```
#define EFI_HII_PACKAGE_TYPE_ALL      0x00
#define EFI_HII_PACKAGE_TYPE_GUID      0x01
#define EFI_HII_PACKAGE_FORMS        0x02
#define EFI_HII_PACKAGE_STRINGS       0x04
#define EFI_HII_PACKAGE_FONTS        0x05
#define EFI_HII_PACKAGE_IMAGES       0x06
#define EFI_HII_PACKAGE_SIMPLE_FONTS    0x07
#define EFI_HII_PACKAGE_DEVICE_PATH     0x08
#define EFI_HII_PACKAGE_KEYBOARD_LAYOUT   0x09
#define EFI_HII_PACKAGE_ANIMATIONS     0x0A
#define EFI_HII_PACKAGE_END         0xDF
#define EFI_HII_PACKAGE_TYPE_SYSTEM_BEGIN  0xE0
#define EFI_HII_PACKAGE_TYPE_SYSTEM_END   0xFF
```

Table 33-8 Package Types

<table>
<thead>
<tr>
<th>Package Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>EFI_HII_PACKAGE_TYPE_ALL</td>
<td>Pseudo-package type used when exporting package lists. See <code>ExportPackageList()</code></td>
</tr>
<tr>
<td>EFI_HII_PACKAGE_TYPE_GUID</td>
<td>Package type where the format of the data is specified using a GUID immediately following the package header.</td>
</tr>
<tr>
<td>EFI_HII_PACKAGE_FORMS</td>
<td>Forms package.</td>
</tr>
<tr>
<td>EFI_HII_PACKAGE_STRINGS</td>
<td>Strings package.</td>
</tr>
<tr>
<td>EFI_HII_PACKAGE_FONTS</td>
<td>Fonts package.</td>
</tr>
<tr>
<td>EFI_HII_PACKAGE_IMAGES</td>
<td>Images package.</td>
</tr>
<tr>
<td>EFI_HII_PACKAGE_SIMPLE_FONTS</td>
<td>Simplified {8x19, 16x19} Fonts package</td>
</tr>
<tr>
<td>EFI_HII_PACKAGE_DEVICE_PATH</td>
<td>Binary-encoded device path.</td>
</tr>
<tr>
<td>EFI_HII_PACKAGE_END</td>
<td>Used to mark the end of a package list.</td>
</tr>
<tr>
<td>EFI_HII_PACKAGE_ANIMATIONS</td>
<td>Animations package.</td>
</tr>
<tr>
<td>EFI_HII_PACKAGE_TYPE_SYSTEM_BEGIN...</td>
<td>Package types reserved for use by platform firmware implementations.</td>
</tr>
<tr>
<td>EFI_HII_PACKAGE_TYPE_SYSTEM_END</td>
<td></td>
</tr>
</tbody>
</table>

### 33.3.1.1 EFI_HII_PACKAGE_LIST_HEADER

**Summary**

The header found at the start of each package list.
Prototype

typedef struct {
    EFI_GUID PackageListGuid;
    UINT32   PackagLength;
} EFI_HII_PACKAGE_LIST_HEADER;

Members

    PackageListGuid   The unique identifier applied to the list of packages which follows.
    PackageLength     The size of the package list (in bytes), including the header.

Description

This header uniquely identifies the package list and is placed in front of a list of packages. Package lists with the same PackageListGuid value should contain the same data set. Updated versions should have updated GUIDs.

33.3.2 Simplified Font Package

The simplified font package describes the font glyphs for the standard 8x19 pixel (narrow) and 16x19 (wide) fonts. Other fonts should be described using the normal Font Package.

A simplified font package consists of a header and two types of glyph structures—standard-width (narrow) and wide glyphs.

33.3.2.1 EFI_HII_SIMPLE_FONT_PACKAGE_HDR

Summary

A simplified font package consists of a font header followed by a series of glyph structures.

Prototype

typedef struct _EFI_HII_SIMPLE_FONT_PACKAGE_HDR {
    EFI_HII_PACKAGE_HEADER    Header;
    UINT16   NumberOfNarrowGlyphs;
    UINT16   NumberOfWideGlyphs;
    EFI_NARROW_GLYPH           NarrowGlyphs[];
    EFI_WIDE_GLYPH             WideGlyphs[];
} EFI_HII_SIMPLE_FONT_PACKAGE_HDR;

Members

    Header

    The header contains a Length and Type field. In the case of a font package, the type will be EFI_HII_PACKAGE_SIMPLE_FONTS and the length will be the total size of the font package including the size of the narrow and wide glyphs. See EFI_HII_PACKAGE_HEADER.

    NumberOfNarrowGlyphs The number of NarrowGlyphs that are included in the font package.
NumberOfWideGlyphs: The number of WideGlyphs that are included in the font package.

NarrowGlyphs: An array of EFI_NARROW_GLYPH entries. The number of entries is specified by NumberOfNarrowGlyphs.

WideGlyphs: An array of EFI_WIDE_GLYPH entries. The number of entries is specified by NumberOfWideGlyphs. To calculate the offset of WideGlyphs, use the offset of NarrowGlyphs and add the size of EFI_NARROW_GLYPH multiplied by the NumberOfNarrowGlyphs.

Description

The glyphs must be sorted by Unicode character code.

It is up to developers who manage fonts to choose efficient mechanisms for accessing fonts. The contiguous presentation can easily be used because narrow and wide glyphs are not intermixed, so a binary search is possible (hence the requirement that the glyphs be sorted by weight).

33.3.2.2 EFI_NARROW_GLYPH

Summary

The EFI_NARROW_GLYPH has a preferred dimension (w x h) of 8 x 19 pixels.

Prototype

typedef struct {
    CHAR16   UnicodeWeight;
    UINT8    Attributes;
    UINT8    GlyphCol1[EFI_GLYPH_HEIGHT];
} EFI_NARROW_GLYPH;

Members

UnicodeWeight: The Unicode representation of the glyph. The term weight is the technical term for a character code.

Attributes: The data element containing the glyph definitions; see "Related Definitions" below.

GlyphCol1: The column major glyph representation of the character. Bits with values of one indicate that the corresponding pixel is to be on when normally displayed; those with zero are off.

Description

Glyphs are represented by two structures, one each for the two sizes of glyphs. The narrow glyph (EFI_NARROW_GLYPH) is the normal glyph used for text display.
Related Definitions

```c
// Contents of EFI_NARROW_GLYPH.Attributes
#define EFI_GLYPH_NON_SPACING 0x01
#define EFI_GLYPH_WIDE 0x02
#define EFI_GLYPH_HEIGHT 19
#define EFI_GLYPH_WIDTH 8
```

Following is a description of the fields in the above definition:

<table>
<thead>
<tr>
<th>Field</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>EFI_GLYPH_NON_SPACING</td>
<td>This symbol is to be printed &quot;on top of&quot; (OR'd with) the previous glyph before display.</td>
</tr>
<tr>
<td>EFI_GLYPH_WIDE</td>
<td>This symbol uses 16x19 formats rather than 8x19.</td>
</tr>
</tbody>
</table>

### 33.3.2.3 EFI_WIDE_GLYPH

**Summary**

The **EFI_WIDE_GLYPH** has a preferred dimension \((w \times h)\) of 16 x 19 pixels, which is large enough to accommodate logographic characters.

**Prototype**

```c
typedef struct {
    CHAR16  UnicodeWeight;
    UINT8   Attributes;
    UINT8   GlyphCol1[EFI_GLYPH_HEIGHT];
    UINT8   GlyphCol2[EFI_GLYPH_HEIGHT];
    UINT8   Pad[3];
} EFI_WIDE_GLYPH;
```

**Members**

- **UnicodeWeight** The Unicode representation of the glyph. The term weight is the technical term for a character code.
- **Attributes** The data element containing the glyph definitions; see "Related Definitions" in **EFI_NARROW_GLYPH** for attribute values.
- **GlyphCol1** and **GlyphCol2** The column major glyph representation of the character. Bits with values of one indicate that the corresponding pixel is to be on when normally displayed; those with zero are off.
- **Pad** Ensures that `sizeof (EFI_WIDE_GLYPH)` is twice the `sizeof (EFI_NARROW_GLYPH)`. The contents of Pad must be zero.

**Description**

Glyphs are represented via the two structures, one each for the two sizes of glyphs. The wide glyph (**EFI_WIDE_GLYPH**) is large enough to display logographic characters.
### 33.3.3 Font Package

The font package describes the glyphs for a single font with a single family, size and style. The package has two parts: a fixed header and the glyph blocks. All structures described here are byte packed.

#### 33.3.3.1 Fixed Header

The fixed header consists of a standard record header and then the character values in this section, the flags (including the encoding method) and the offsets of the glyph information, the glyph bitmaps and the character map.

```c
typedef struct _EFI_HII_FONT_PACKAGE_HDR {
  EFI_HII_PACKAGE_HEADER Header;
  UINT32 HdrSize;
  UINT32 GlyphBlockOffset;
  EFI_HII_GLYPH_INFO Cell;
  EFI_HII_FONT_STYLE FontStyle;
  CHAR16 FontFamily[];
} EFI_HII_FONT_PACKAGE_HDR;
```

- **Header** The standard package header, where `Header.Type = EFI_HII_PACKAGE_FONTS`.
- **HdrSize** Size of this header.
- **GlyphBlockOffset** The offset, relative to the start of this header, of a series of variable-length glyph blocks, each describing information about the bitmap associated with a glyph.
- **Cell** This contains the measurement of the widest and tallest characters in the font (`Cell.Width` and `Cell.Height`). It also contains the default offset to the horizontal and vertical origin point of the character cell (`Cell.OffsetX` and `Cell.OffsetY`). Finally, it contains the default `AdvanceX`.
- **FontStyle** The design style of the font, 1 bit per style. See `EFI_HII_FONT_STYLE`.
- **FontFamily** The null-terminated string with the name of the font family to which the font belongs.
Related Definitions

```c
typedef UINT32 EFI_HII_FONT_STYLE;
#define EFI_HII_FONT_STYLE_NORMAL    0x00000000
#define EFI_HII_FONT_STYLE_BOLD     0x00000001
#define EFI_HII_FONT_STYLE_ITALIC    0x00000002
#define EFI_HII_FONT_STYLE_EMBOSSED 0x00010000
#define EFI_HII_FONT_STYLE_OUTLINE  0x00020000
#define EFI_HII_FONT_STYLE_SHADOW    0x00040000
#define EFI_HII_FONT_STYLE_UNDERLINE 0x00080000
#define EFI_HII_FONT_STYLE_DBL_UNDER 0x00100000
```

### 33.3.3.2 Glyph Information

For each Unicode character code, the glyph information gives the glyph bitmap, the character size and the position of the bitmap relative to the origin of the character cell. The glyph information is encoded as a series of blocks, each with a single byte header. The blocks must be processed in order.

Each block begins with a single byte, which contains the block type.

![Figure 33-36 Glyph Information Encoded in Blocks](image)
Prototype

```c
typedef struct _EFI_HII_GLYPH_BLOCK {
    UINT8 BlockType;
    UINT8 BlockBody[];
} EFI_HII_GLYPH_BLOCK;
```

Members

The following table describes the different block types:

<table>
<thead>
<tr>
<th>Name</th>
<th>Value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>EFI_HII_GIBT_END</td>
<td>0x00</td>
<td>The end of the glyph information.</td>
</tr>
<tr>
<td>EFI_HII_GIBT_GLYPH</td>
<td>0x10</td>
<td>Glyph information for a single character value, bit-packed.</td>
</tr>
<tr>
<td>EFI_HII_GIBT_GLYPHS</td>
<td>0x11</td>
<td>Glyph information for multiple character values.</td>
</tr>
<tr>
<td>EFI_HII_GIBT_GLYPH_DEFAULT</td>
<td>0x12</td>
<td>Glyph information for a single character value, using the default character cell information.</td>
</tr>
<tr>
<td>EFI_HII_GIBT_GLYPHS_DEFAULT</td>
<td>0x13</td>
<td>Glyph information for multiple character values, using the default character cell information.</td>
</tr>
<tr>
<td>EFI_HII_GIBT_GLYPH_VARIABILITY</td>
<td>0x14</td>
<td>Glyph information for the variable glyph.</td>
</tr>
<tr>
<td>EFI_HII_GIBT_DUPLICATE</td>
<td>0x20</td>
<td>Create a duplicate of an existing glyph but with a new character value.</td>
</tr>
<tr>
<td>EFI_HII_GIBT_SKIP2</td>
<td>0x21</td>
<td>Skip a number (1-65535) character values.</td>
</tr>
<tr>
<td>EFI_HII_GIBT_SKIP1</td>
<td>0x22</td>
<td>Skip a number (1-255) character values.</td>
</tr>
<tr>
<td>EFI_HII_GIBT_DEFAULTS</td>
<td>0x23</td>
<td>Set default glyph information for subsequent glyph blocks.</td>
</tr>
<tr>
<td>EFI_HII_GIBT_EXT1</td>
<td>0x30</td>
<td>For future expansion (one byte length field)</td>
</tr>
<tr>
<td>EFI_HII_GIBT_EXT2</td>
<td>0x31</td>
<td>For future expansion (two byte length field)</td>
</tr>
<tr>
<td>EFI_HII_GIBT_EXT4</td>
<td>0x32</td>
<td>For future expansion (four byte length field)</td>
</tr>
</tbody>
</table>

Description

In order to recreate all glyphs, start at the first block and process them all until a **EFI_HII_GIBT_END** block is found. When processing the glyph blocks, each block refers to the current character value (**CharValueCurrent**), which is initially set to one (1).

Glyph blocks of an unknown type should be skipped. If they cannot be skipped, then processing halts.
Related Definitions

```c
typedef struct _EFI_HII_GLYPH_INFO {
    UINT16 Width;
    UINT16 Height;
    INT16 OffsetX;
    INT16 OffsetY;
    INT16 AdvanceX;
} EFI_HII_GLYPH_INFO;
```

- **Width**: Width of the character or character cell, in pixels. For fixed-pitch fonts, this is the same as the advance.
- **Height**: Height of the character or character cell, in pixels.
- **OffsetX**: Offset to the horizontal edge of the character cell.
- **OffsetY**: Offset to the vertical edge of the character cell.
- **AdvanceX**: Number of pixels to advance to the right when moving from the origin of the current glyph to the origin of the next glyph.

### 33.3.3.2.1 EFI_HII_GIBT_DEFAULTS

**Summary**

Changes the default character cell information.

**Prototype**

```c
typedef struct _EFI_HII_GIBT_DEFAULTS_BLOCK {
    EFI_HII_GLYPH_BLOCK Header;
    EFI_HII_GLYPH_INFO Cell;
} EFI_HII_GIBT_DEFAULTS_BLOCK;
```

**Members**

- **Header**: Standard glyph block header, where `Header.BlockType = EFI_HII_GIBT_DEFAULTS`.
- **Cell**: The new default cell information which will be applied to all subsequent `GLYPH_DEFAULT` and `GLYPHS_DEFAULT` blocks.

**Description**

Changes the default cell information used for subsequent `EFI_HII_GIBT_GLYPH_DEFAULT` and `EFI_HII_GIBT_GLYPHS_DEFAULT` glyph blocks. The cell information described by `Cell` remains in effect until the next `EFI_HII_GIBT_DEFAULTS` is found. Prior to the first `EFI_HII_GIBT_DEFAULTS` block, the cell information in the fixed header are used.
33.3.3.2.2 EFI_HII_GIBT_DUPLICATE

Summary
Assigns a new character value to a previously defined glyph.

Prototype
```c
typedef struct _EFI_HII_GIBT_DUPLICATE_BLOCK {
    EFI_HII_GLYPH_BLOCK    Header;
    CHAR16                 CharValue;
} EFI_HII_GIBT_DUPLICATE_BLOCK;
```

Members
- **Header**
  Standard glyph block header, where `Header.BlockType = EFI_HII_GIBT_DUPLICATE`.
- **CharValue**
  The previously defined character value with the exact same glyph.

Description
Indicates that the glyph with character value `CharValueCurrent` has the same glyph as a previously defined character value and increments `CharValueCurrent` by one.

33.3.3.2.3 EFI_HII_GIBT_END

Summary
Marks the end of the glyph information.

Prototype
```c
typedef struct _EFI_GLYPH_GIBT_END_BLOCK {
    EFI_HII_GLYPH_BLOCK    Header;
} EFI_GLYPH_GIBT_END_BLOCK;
```

Members
- **Header**
  Standard glyph block header, where `Header.BlockType = EFI_HII_GIBT_END`.

Description
Any glyphs with a character value greater than or equal to `CharValueCurrent` are empty.

33.3.3.2.4 EFI_HII_GIBT_EXT1, EFI_HII_GIBT_EXT2, EFI_HII_GIBT_EXT4

Summary
Future expansion block types which have a length byte.
Prototype

```c
typedef struct _EFI_HII_GIBT_EXT1_BLOCK {
    EFI_HII_GLYPH_BLOCK Header;
    UINT8     BlockType2;
    UINT8     Length;
} EFI_HII_GIBT_EXT1_BLOCK;

typedef struct _EFI_HII_GIBT_EXT2_BLOCK {
    EFI_HII_GLYPH_BLOCK Header;
    UINT8     BlockType2;
    UINT16    Length;
} EFI_HII_GIBT_EXT2_BLOCK;

typedef struct _EFI_HII_GIBT_EXT4_BLOCK {
    EFI_HII_GLYPH_BLOCK Header;
    UINT8     BlockType2;
    UINT32    Length;
} EFI_HII_GIBT_EXT4_BLOCK;
```

Members

- **Header**
  - Standard glyph block header, where `Header.BlockType` = `EFI_HII_GIBT_EXT1`, `EFI_HII_GIBT_EXT2` or `EFI_HII_GIBT_EXT4`.

- **Length**
  - Size of the glyph block, in bytes.

- **BlockType2**
  - Indicates the type of extended block. Currently all extended block types are reserved for future expansion.

Description

These are reserved for future expansion, with length bytes included so that they can be easily skipped.

**33.3.3.2.5 EFI_HII_GIBT_GLYPH**

Summary

Provide the bitmap for a single glyph.
Prototype

```c
typedef struct _EFI_HII_GIBT_GLYPH_BLOCK {
    EFI_HII_GLYPH_BLOCK Header;
    EFI_HII_GLYPH_INFO Cell;
    UINT8 BitmapData[1];
} EFI_HII_GIBT_GLYPH_BLOCK;
```

Members

- **Header**
  Standard glyph block header, where `Header.BlockType` = `EFI_HII_GIBT_GLYPH`.

- **Cell**
  Contains the width and height of the encoded bitmap (`Cell.Width` and `Cell.Height`), the number of pixels (signed) right of the character cell origin where the left edge of the bitmap should be placed (`Cell.OffsetX`), the number of pixels above the character cell origin where the top edge of the bitmap should be placed (`Cell.OffsetY`) and the number of pixels (signed) to move right to find the origin for the next character cell (`Cell.AdvanceX`).

- **GlyphCount**
  The number of glyph bitmaps.

- **BitmapData**
  The bitmap data specifies a series of pixels, one bit per pixel, left-to-right, top-to-bottom. Each glyph bitmap only encodes the portion of the bitmap enclosed by its character-bounding box, but the entire glyph is padded out to the nearest byte. The number of bytes per bitmap can be calculated as: `((Cell.Width + 7)/8) * Cell.Height`.

Description

This block provides the bitmap for the character with the value `CharValueCurrent` and increments `CharValueCurrent` by one. Each glyph contains a glyph width and height, a drawing offset, number of pixels to advance after drawing and then the encoded bitmap.

33.3.3.2.6 EFI_HII_GIBT_GLYPHS

Summary

Provide the bitmaps for multiple glyphs with the same cell information.
Prototype

typedef struct _EFI_HII_GIBT_GLYPHS_BLOCK {
    EFI_HII_GLYPH_BLOCK Header;
    EFI_HII_GLYPH_INFO Cell;
    UINT16 Count
    UINT8 BitmapData[1];
} EFI_HII_GIBT_GLYPHS_BLOCK;

Members

Header

Standard glyph block header, where Header.BlockType = EFI_HII_GIBT_GLYPHS.

Cell

Contains the width and height of the encoded bitmap (Cell.Width and Cell.Height), the number of pixels (signed) right of the character cell origin where the left edge of the bitmap should be placed (Cell.OffsetX), the number of pixels above the character cell origin where the top edge of the bitmap should be placed (Cell.OffsetY) and the number of pixels (signed) to move right to find the origin for the next character cell (Cell.AdvanceX).

BitmapData

The bitmap data specifies a series of pixels, one per pixel, left-to-right, top-to-bottom, for each glyph. Each glyph bitmap only encodes the portion of the bitmap enclosed by its character-bounding box. The number of bytes per bitmap can be calculated as: \((\text{Cell.Width} + 7)/8\) * Cell.Height.

Description

Provides the bitmaps for the characters with the values CharValueCurrent through CharValueCurrent + Count - 1 and increments CharValueCurrent by Count. These glyphs have identical cell information and the encoded bitmaps are exactly the same number of bytes.

33.3.3.2.7 EFI_HII_GIBT_GLYPH_DEFAULT

Summary

Provide the bitmap for a single glyph, using the default cell information.

Prototype

typedef struct _EFI_HII_GIBT_GLYPH_DEFAULT_BLOCK {
    EFI_HII_GLYPH_BLOCK Header;
    UINT8 BitmapData[];
} EFI_HII_GIBT_GLYPH_DEFAULT_BLOCK;

Members

Header

Standard glyph block header, where Header.BlockType = EFI_HII_GIBT_GLYPH_DEFAULT.
**BitmapData**

The bitmap data specifies a series of pixels, one bit per pixel, left-to-right, top-to-bottom. Each glyph bitmap only encodes the portion of the bitmap enclosed by its character-bounding box. The number of bytes per bitmap can be calculated as:

\[
((\text{Global.Cell.Width} + 7)/8) \times \text{Global.Cell.Height}
\]

**Description**

Provides the bitmap for the character with the value `CharValueCurrent` and increments `CharValueCurrent` by 1. This glyph uses the default cell information. The default cell information is found in the font header or the most recently processed `EFI_HII_GIBT_DEFAULTS`.

### 33.3.3.2.8 EFI_HII_GIBT_GLYPHS_DEFAULT

**Summary**

Provide the bitmaps for multiple glyphs with the default cell information

**Prototype**

```c
typedef struct _EFI_HII_GIBT_GLYPHS_DEFAULT_BLOCK {
    EFI_HII_GLYPH_BLOCK Header;
    UINT16       Count;
    UINT8        BitmapData[];
} EFI_HII_GIBT_GLYPHS_DEFAULT_BLOCK;
```

**Members**

- **Header**
  
  Standard glyph block header, where `Header.BlockType = EFI_HII_GIBT_GLYPHS_DEFAULT`.

- **Count**
  
  Number of glyphs in the glyph block.

- **BitmapData**
  
  The bitmap data specifies a series of pixels, one bit per pixel, left-to-right, top-to-bottom, for each glyph. Each glyph bitmap only encodes the portion of the bitmap enclosed by its character-bounding box. The number of bytes per bitmap can be calculated as:

  \[
  ((\text{Global.Cell.Width} + 7)/8) \times \text{Global.Cell.Height}
  \]

**Description**

Provides the bitmaps for the characters with the values `CharValueCurrent` through `CharValueCurrent + Count - 1` and increments `CharValueCurrent` by `Count`. These glyphs use the default cell information and the encoded bitmaps have exactly the same number of bytes.

### 33.3.3.2.9 EFI_HII_GIBT_SKIPx

**Summary**

Increments the current character value `CharValueCurrent` by the number specified.
Prototype

```c
typedef struct _EFI_HII_GIBT_SKIP2_BLOCK {
    EFI_HII_GLYPH_BLOCK  Header;
    UINT16               SkipCount;
} EFI_HII_GIBT_SKIP2_BLOCK;

typedef struct _EFI_HII_GIBT_SKIP1_BLOCK {
    EFI_HII_GLYPH_BLOCK  Header;
    UINT8               SkipCount;
} EFI_HII_GIBT_SKIP1_BLOCK;
```

Members

- **Header**
  Standard glyph block header, where `BlockType = EFI_HII_GIBT_SKIP1` or `EFI_HII_GIBT_SKIP2`.

- **SkipCount**
  The unsigned 8- or 16-bit value to add to `CharValueCurrent`.

Description

Increments the current character value `CharValueCurrent` by the number specified.

### 33.3.3.2.10 EFI_HII_GIBT_GLYPH_VARIABILITY

Related Definitions

```c
// **************************************************************
// EFI_HII_GIBT_GLYPH_VARIABILITY (0x14)
// **************************************************************
typedef struct _EFI_HII_GIBT_VARIABILITY_BLOCK {
    EFI_HII_GLYPH_BLOCK  Header;
    EFI_HII_GLYPH_INFO   Cell;
    UINT8               GlyphPackInBits;
    UINT8               BitmapData [1];
} EFI_HII_GIBT_VARIABILITY_BLOCK;
```

Member

- **Header**
  Standard glyph block header, where `BlockType = EFI_HII_GIBT_GLYPH_VARIABILITY`.

- **Cell**
  Contains the width and height of the encoded bitmap (`Cell.Width` and `Cell.Height`), the number of pixels (signed) right of the character cell origin where the left edge of the bitmap should be placed (`Cell.OffsetX`), the number of pixels above the character cell origin where the top edge of the
bitmap should be placed (Cell.OffsetY) and the number of pixels (signed) to move right to find the origin for the next character cell (Cell.AdvanceX).

**GlyphPackInBits**

This describes the bit length for each pixel in glyph. With this, the length of BitmapData can be determined according to GlyphPackInBits, cell.with and cell.height.

The valid value is `GIBT_VARIABILITY_BLOCK_1_BIT`, `GIBT_VARIABILITY_BLOCK_2_BIT`, `GIBT_VARIABILITY_BLOCK_4_BIT`, `GIBT_VARIABILITY_BLOCK_8_BIT`, `GIBT_VARIABILITY_BLOCK_16_BIT`, `GIBT_VARIABILITY_BLOCK_24_BIT`, `GIBT_VARIABILITY_BLOCK_32_BIT`

HII Font Ex protocol has no idea about how to decode the bitmap of glyph if the glyph is declared as `EFI_HII_GIBT_GLYPH_VARIABILITY`. The bitmap decoding is resolved in `EFI_HII_FONT_GLPHY_GENERATOR_PROTOCOL`. This field is used to determine the length of entire glyph block.

**BitmapData**

The raw data of the glyph pixels. The format of the glyph pixel depends on the glyph generator. Only `EFI_HII_FONT_GLPHY_GENERATOR_PROTOCOL` knows how to draw the glyph.
33.3.4 Device Path Package

Summary
The device path package is used to carry a device path associated with the package list.

Prototype

```c
typedef struct _EFI_HII_DEVICE_PATH_PACKAGE {
    EFI_HII_PACKAGE_HEADER Header;
    EFI_DEVICE_PATH_PROTOCOL DevicePath[];
} EFI_HII_DEVICE_PATH_PACKAGE;
```

Parameters

- **Header**
  The standard package header, where `Header.Type = EFI_HII_PACKAGE_DEVICE_PATH`.

- **DevicePath**
  The Device Path description associated with the driver handle that provided the content sent to the HII database.
### Description

This package is created by `NewPackageList()` when the package list is first added to the HII database by locating the ` EFI_DEVICE_PATH_PROTOCOL` attached to the driver handle passed in to that function.

### 33.3.5 GUID Package

The GUID package is used to carry data where the format is defined by a GUID.

#### Prototype

```c
typedef struct _EFI_HII_GUID_PACKAGE_HDR {
    EFI_HII_PACKAGE_HEADER Header;
    EFI_GUID Guid;
    // Data per GUID definition may follow
} EFI_HII_GUID_PACKAGE_HDR;
```

#### Members

- **Header**
  - The standard package header, where `Header.Type` = `EFI_HII_PACKAGE_TYPE_GUID`.

- **Guid**
  - Identifier which describes the remaining data within the package.

#### Description

This is a free-form package type designed to allow extensibility by allowing the format to be specified using `Guid`.

### 33.3.6 String Package

The Strings package record describes the mapping between string identifiers and the actual text of the strings themselves. The package consists of three parts: a fixed header, the string information and the font information.

#### 33.3.6.1 Fixed Header

The fixed header consists of a standard record header and then the string identifiers contained in this section and the offsets of the string and language information.

#### Prototype

```c
typedef struct _EFI_HII_STRING_PACKAGE_HDR {
    EFI_HII_PACKAGE_HEADER Header;
    UINT32 HdrSize;
    UINT32 StringInfoOffset;
    CHAR16 LanguageWindow[16];
    EFI_STRING_ID LanguageName;
    CHAR8 Language[ ... ];
} EFI_HII_STRING_PACKAGE_HDR;
```
Members

Header

The standard package header, where Header.Type = EFI_HII_PACKAGE_STRINGS.

HdrSize

Size of this header.

StringInfoOffset

Offset, relative to the start of this header, of the string information.

LanguageWindow

Specifies the default values placed in the static and dynamic windows before processing each SCSU-encoded string.

LanguageName

String identifier within the current string package of the full name of the language specified by Language.

Language

The null-terminated ASCII string that specifies the language of the strings in the package. The languages are described as specified by Appendix M.

Related Definition

#define UEFI_CONFIG_LANG "x-UEFI"
#define UEFI_CONFIG_LANG_2 "x-i-UEFI"

33.3.6.2 String Information

For each string identifier, the string information gives the string’s text and font. The string information is encoded as a series of blocks, each with a single byte header. The blocks must be processed in order, using the current string identifier (StringIdCurrent), which is set initially to one (1). Processing continues until an EFI_SIBT_END block is found.

The types of blocks are: string blocks, duplicate blocks, font blocks, and skip blocks. String blocks specify the text and font for the current string identifier and increment to the next string identifier. Duplicate blocks copy the text of a previous string identifier and increment to the next string identifier. Skip blocks skip string identifiers, leaving them blank.
Figure 33-39 String Information Encoded in Blocks

Each block begins with a single byte, which contains the block type.

```c
typedef struct {
    UINT8 BlockType;
    UINT8 BlockBody[];
} EFI_HII_STRING_BLOCK;
```

The following table describes the different block types:

<table>
<thead>
<tr>
<th>Name</th>
<th>Value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>EFI_HII_SIBT_END</td>
<td>0x00</td>
<td>The end of the string information.</td>
</tr>
<tr>
<td>EFI_HII_SIBT_STRING_SCSU</td>
<td>0x10</td>
<td>Single string using default font information.</td>
</tr>
<tr>
<td>EFI_HII_SIBT_STRING_SCSU_FONT</td>
<td>0x11</td>
<td>Single string with font information.</td>
</tr>
<tr>
<td>EFI_HII_SIBT_STRINGS_SCSU</td>
<td>0x12</td>
<td>Multiple strings using default font information.</td>
</tr>
<tr>
<td>EFI_HII_SIBT_STRINGS_SCSU_FONT</td>
<td>0x13</td>
<td>Multiple strings with font information.</td>
</tr>
<tr>
<td>EFI_HII_SIBT_STRING_UCS2</td>
<td>0x14</td>
<td>Single UCS-2 string using default font information.</td>
</tr>
<tr>
<td>EFI_HII_SIBT_STRING_UCS2_FONT</td>
<td>0x15</td>
<td>Single UCS-2 string with font information</td>
</tr>
<tr>
<td>EFI_HII_SIBT_STRINGS_UCS2</td>
<td>0x16</td>
<td>Multiple UCS-2 strings using default font information.</td>
</tr>
</tbody>
</table>
When processing the string blocks, each block type refers and modifies the current string identifier (StringIdCurrent).

<table>
<thead>
<tr>
<th>String Type</th>
<th>Value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>EFI_HII_SIBT_STRINGS_UCS2_FONT</td>
<td>0x17</td>
<td>Multiple UCS-2 strings with font information.</td>
</tr>
<tr>
<td>EFI_HII_SIBT_DUPLICATE</td>
<td>0x20</td>
<td>Create a duplicate of an existing string.</td>
</tr>
<tr>
<td>EFI_HII_SIBT_SKIP2</td>
<td>0x21</td>
<td>Skip a certain number of string identifiers.</td>
</tr>
<tr>
<td>EFI_HII_SIBT_SKIP1</td>
<td>0x22</td>
<td>Skip a certain number of string identifiers.</td>
</tr>
<tr>
<td>EFI_HII_SIBT_EXT1</td>
<td>0x30</td>
<td>For future expansion (one byte length field)</td>
</tr>
<tr>
<td>EFI_HII_SIBT_EXT2</td>
<td>0x31</td>
<td>For future expansion (two byte length field)</td>
</tr>
<tr>
<td>EFI_HII_SIBT_EXT4</td>
<td>0x32</td>
<td>For future expansion (four byte length field)</td>
</tr>
<tr>
<td>EFI_HII_SIBT_FONT</td>
<td>0x40</td>
<td>Font information.</td>
</tr>
</tbody>
</table>
Current = 1

A

B
No

StringBlock BlockType = DUPLICATE?
Yes

String[Current] = String[StringBlock, StringId]
StringCount = 1

C

Advance To Next Block

Yes

StringCount = BlockValue

No

Current += StringCount

StringBlock BlockType = SKIPx?
Yes

No

StringBlock BlockType = EXTx?
Yes

No

StringBlock BlockType = FONT
Yes

Font[StringBlock.FontId] = StringBlock.FontInfo

No

Return BlockType = END?
Figure 33-41 String Block Processing: SCSU Processing
33.3.6.2.1 EFI_HII_SIBT_DUPLICATE

Summary
Creates a duplicate of a previously defined string.

Prototype
```c
typedef struct _EFI_HII_SIBT_DUPLICATE_BLOCK {
  EFI_HII_STRING_BLOCK     Header;
  EFI_STRING_ID            StringId;
} EFI_HII_SIBT_DUPLICATE_BLOCK;
```
Members

Header

Standard string block header, where Header.BlockType = EFI_HII_SIBT_DUPLICATE.

StringId

The string identifier of a previously defined string with the exact same string text.

Description

Indicates that the string with string identifier StringIdCurrent is the same as a previously defined string and increments StringIdCurrent by one.

33.3.6.2.2 EFI_HII_SIBT_END

Summary

Marks the end of the string information.

Prototype

typedef struct _EFI_HII_SIBT_END_BLOCK {
  EFI_HII_STRING_BLOCK Header;
} EFI_HII_SIBT_END_BLOCK;

Members

Header

Standard extended header, where Header.Header.BlockType = EFI_HII_SIBT_EXT2 and Header.BlockType2 = EFI_HII_SIBT_FONT.

BlockType2

Indicates the type of extended block. See Section 33.3.6.2 for a list of all block types.

Description

Any strings with a string identifier greater than or equal to StringIdCurrent are empty.

33.3.6.2.3 EFI_HII_SIBT_EXT1, EFI_HII_SIBT_EXT2, EFI_HII_SIBT_EXT4

Summary

Future expansion block types which have a length byte.
Prototype

```
typedef struct _EFI_HII_SIBT_EXT1_BLOCK {
    EFI_HII_STRING_BLOCK  Header;
    UINT8 BlockType2;
    UINT8 Length;
} EFI_HII_SIBT_EXT1_BLOCK;

typedef struct _EFI_HII_SIBT_EXT2_BLOCK {
    EFI_HII_STRING_BLOCK  Header;
    UINT8 BlockType2;
    UINT16 Length;
} EFI_HII_SIBT_EXT2_BLOCK;

typedef struct _EFI_HII_SIBT_EXT4_BLOCK {
    EFI_HII_STRING_BLOCK  Header;
    UINT8 BlockType2;
    UINT32 Length;
} EFI_HII_SIBT_EXT4_BLOCK;
```

Members

- **Header**
  Standard string block header, where `Header.BlockType = EFI_HII_SIBT_EXT1`, `EFI_HII_SIBT_EXT2` or `EFI_HII_SIBT_EXT4`.

- **Length**
  Size of the string block, in bytes.

- **BlockType2**
  Indicates the type of extended block. See Section 33.3.6.2 for a list of all block types.

Description

These are reserved for future expansion, with length bytes included so that they can be easily skipped.

### 33.3.6.2.4 EFI_HII_SIBT_FONT

Summary

Provide information about a single font.
Prototype

typedef struct _EFI_HII_SIBT_FONT_BLOCK {
    EFI_HII_SIBT_EXT2_BLOCK Header;
    UINT8 FontId;
    UINT16 FontSize;
    EFI_HII_FONT_STYLE FontStyle;
    CHAR16 FontName[...];
} EFI_HII_SIBT_FONT_BLOCK;

Members

  Header
    Standard extended header, where Header.BlockType2 = EFI_HII_SIBT_FONT.

  FontId
    Font identifier, which must be unique within the string package.

  FontSize
    Character cell size, in pixels, of the font.

  FontStyle
    Font style. Type EFI_HII_FONT_STYLE is defined in “Related Definitions” in EFI_HII_FONT_PACKAGE_HDR.

  FontName
    Null-terminated font family name.

Description

Associates a font identifier FontId with a font name FontName, size FontSize and style FontStyle. This font identifier may be used with the string blocks. The font identifier 0 is the default font for those string blocks which do not specify a font identifier.

33.3.6.2.5 EFI_HII_SIBT_SKIP1

Summary

Skips string identifiers.

Prototype

typedef struct _EFI_HII_SIBT_SKIP1_BLOCK {
    EFI_HII_STRING_BLOCK Header;
    UINT8 SkipCount;
} EFI_HII_SIBT_SKIP1_BLOCK;

Members

  Header
    Standard string block header, where Header.BlockType = EFI_HII_SIBT_SKIP1.
SkipCount

The unsigned 8-bit value to add to StringIdCurrent.

Description
Increments the current string identifier StringIdCurrent by the number specified.

33.3.6.2.6 EFI_HII_SIBT_SKIP2

Summary
Skips string ids.

Prototype

typedef struct _EFI_HII_SIBT_SKIP2_BLOCK {
    EFI_HII_STRING_BLOCK       Header;
    UINT16                     SkipCount;
} EFI_HII_SIBT_SKIP2_BLOCK;

Members

Header
Standard string block header, where Header.BlockType = EFI_HII_SIBT_SKIP2.

SkipCount
The unsigned 16-bit value to add to StringIdCurrent.

Description
Increments the current string identifier StringIdCurrent by the number specified.

33.3.6.2.7 EFI_HII_SIBT_STRING_SCSU

Summary
Describe a string encoded using SCSU, in the default font.

Prototype

typedef struct _EFI_HII_SIBT_STRING_SCSU_BLOCK {
    EFI_HII_STRING_BLOCK     Header;
    UINT8                    StringText[];
} EFI_HII_SIBT_STRING_SCSU_BLOCK;

Members

Header
Standard header where Header.BlockType = EFI_HII_SIBT_STRING_SCSU.

StringText
The string text is a null-terminated string, which is assigned to the string identifier StringIdCurrent.
Description
This string block provides the SCSU-encoded text for the string in the default font with string identifier StringIdCurrent and increments StringIdCurrent by one.

33.3.6.2.8 EFI_HII_SIBT_STRING_SCSU_FONT

Summary
Describe a string in the specified font.

Prototype

typedef struct _EFI_HII_SIBT_STRING_SCSU_FONT_BLOCK {
  EFI_HII_STRING_BLOCK Header;
  UINT8 FontIdentifier;
  UINT8 StringText[];
} EFI_HII_SIBT_STRING_SCSU_FONT_BLOCK;

Members

  Header
  Standard string block header, where Header.BlockType = EFI_HII_SIBT_STRING_SCSU_FONT.

  FontIdentifier
  The identifier of the font to be used as the starting font for the entire string. The identifier must either be 0 for the default font or an identifier previously specified by an EFI_HII_SIBT_FONT block. Any string characters that deviates from this font family, size or style must provide an explicit control character. See Section 33.2.6.2.4.

  StringText
  The string text is a null-terminated encoded string, which is assigned to the string identifier StringIdCurrent.

Description
This string block provides the SCSU-encoded text for the string in the font specified by FontIdentifier with string identifier StringIdCurrent and increments StringIdCurrent by one.

33.3.6.2.9 EFI_HII_SIBT_STRINGS_SCSU

Summary
Describe strings in the default font.
Prototype

```c
typedef struct _EFI_HII_SIBT_STRINGS_SCSU_BLOCK {
  EFI_HII_STRING_BLOCK Header;
  UINT16 StringCount;
  UINT8 StringText[];
} EFI_HII_SIBT_STRINGS_SCSU_BLOCK;
```

Members

- **Header**
  Standard header where `Header.BlockType = EFI_HII_SIBT_STRINGS_SCSU`.

- **StringCount**
  Number of strings in `StringText`.

- **StringText**
  The strings, where each string is a null-terminated encoded string.

Description

This string block provides the SCSU-encoded text for `StringCount` strings which have the default font and which have sequential string identifiers. The strings are assigned the identifiers, starting with `StringIdCurrent` and continuing through `StringIdCurrent + StringCount – 1`. `StringIdCurrent` is incremented by `StringCount`.

33.3.6.2.10 EFI_HII_SIBT_STRINGS_SCSU_FONT

Summary

Describe strings in the specified font.

Prototype

```c
typedef struct _EFI_HII_SIBT_STRINGS_SCSU_FONT_BLOCK {
  EFI_HII_STRING_BLOCK Header;
  UINT8 FontIdentifier;
  UINT16 StringCount;
  UINT8 StringText[];
} EFI_HII_SIBT_STRINGS_SCSU_FONT_BLOCK;
```

Members

- **Header**
  Standard header where `Header.BlockType = EFI_HII_SIBT_STRINGS_SCSU_FONT`.

- **StringCount**
  Number of strings in `StringText`.

- **FontIdentifier**
  The identifier of the font to be used as the starting font for the entire string. The identifier must either be 0 for the default font or an identifier previously specified by
an **EFI_HII_SIBT_FONT** block. Any string characters that deviates from this font family, size or style must provide an explicit control character. See [Section 33.2.6.2.4](#).

**StringText**

The strings, where each string is a null-terminated encoded string.

**Description**

This string block provides the SCSU-encoded text for **StringCount** strings which have the font specified by **FontIdentifier** and which have sequential string identifiers. The strings are assigned the identifiers, starting with **StringIdCurrent** and continuing through **StringIdCurrent + StringCount – 1**. **StringIdCurrent** is incremented by **StringCount**.

### 33.3.6.2.11 EFI_HII_SIBT_STRING_UCS2

**Summary**

Describe a string in the default font.

**Prototype**

```c
typedef struct _EFI_HII_SIBT_STRING_UCS2_BLOCK {
    EFI_HII_STRING_BLOCK     Header;
    CHAR16                   StringText[];
} EFI_HII_SIBT_STRING_UCS2_BLOCK;
```

**Members**

- **Header**
  
  Standard header where **Header.BlockType** = **EFI_HII_SIBT_STRING_UCS2**.

- **StringText**
  
  The string text is a null-terminated UCS-2 string, which is assigned to the string identifier **StringIdCurrent**.

**Description**

This string block provides the UCS-2 encoded text for the string in the default font with string identifier **StringIdCurrent** and increments **StringIdCurrent** by one.

### 33.3.6.2.12 EFI_HII_SIBT_STRING_UCS2_FONT

**Summary**

Describe a string in the specified font.
Prototype

```c
typedef struct _EFI_HII_SIBT_STRING_UCS2_FONT_BLOCK {
  EFI_HII_STRING_BLOCK   Header;
  UINT8                  FontIdentifier;
  CHAR16                 StringText[];
} EFI_HII_SIBT_STRING_UCS2_FONT_BLOCK;
```

Members

- **Header**
  Standard header where `Header.BlockType = EFI_HII_SIBT_STRING_UCS2_FONT`.

- **FontIdentifier**
  The identifier of the font to be used as the starting font for the entire string. The identifier must either be 0 for the default font or an identifier previously specified by an `EFI_HII_SIBT_FONT` block. Any string characters that deviates from this font family, size or style must provide an explicit control character. See Section 33.2.6.2.4.

- **StringText**
  The string text is a null-terminated UCS-2 string, which is assigned to the string identifier `StringIdCurrent`.

Description

This string block provides the UCS-2 encoded text for the string in the font specified by `FontIdentifier` with string identifier `StringIdCurrent` and increments `StringIdCurrent` by one.

33.3.6.2.13 EFI_HII_SIBT_STRINGS_UCS2

Summary

Describes strings in the default font.

Prototype

```c
typedef struct _EFI_HII_SIBT_STRINGS_UCS2_BLOCK {
  EFI_HII_STRING_BLOCK   Header;
  UINT16                 StringCount;
  CHAR16                 StringText[];
} EFI_HII_SIBT_STRINGS_UCS2_BLOCK;
```

Members

- **Header**
  Standard header where `Header.BlockType = EFI_HII_SIBT_STRINGS_UCS2`.

- **StringCount**
  Number of strings in `StringText`.
**StringText**

The string text is a series of null-terminated UCS-2 strings, which are assigned to the string identifiers `StringIdCurrent` to `StringIdCurrent + StringCount – 1`.

**Description**

This string block provides the UCS-2 encoded text for the strings in the default font with string identifiers `StringIdCurrent` to `StringIdCurrent + StringCount – 1` and increments `StringIdCurrent` by `StringCount`.

### 33.3.6.2.14 EFI_HII_SIBT_STRINGS_UCS2_FONT

**Summary**

Describes strings in the specified font.

**Prototype**

```c
typedef struct _EFI_HII_SIBT_STRINGS_UCS2_FONT_BLOCK {
    EFI_HII_STRING_BLOCK    Header;
    UINT8                   FontIdentifier;
    UINT16                  StringCount;
    CHAR16                  StringText[];
} EFI_HII_SIBT_STRINGS_UCS2_FONT_BLOCK;
```

**Members**

* **Header**
  
  Standard header where `Header.BlockType = EFI_HII_SIBT_STRINGS_UCS2_FONT`.

* **FontIdentifier**
  
  The identifier of the font to be used as the starting font for the entire string. The identifier must either be 0 for the default font or an identifier previously specified by an `EFI_HII_SIBT_FONT` block. Any string characters that deviates from this font family, size or style must provide an explicit control character. See Section 33.2.6.2.4.

* **StringCount**

  Number of strings in `StringText`.

* **StringText**

  The string text is a series of null-terminated UCS-2 strings, which are assigned to the string identifiers `StringIdCurrent` through `StringIdCurrent + StringCount – 1`.

**Description**

This string block provides the UCS-2 encoded text for the strings in the font specified by `FontIdentifier` with string identifiers `StringIdCurrent` to `StringIdCurrent + StringCount – 1` and increments `StringIdCurrent` by `StringCount`. 
33.3.6.3 String Encoding

Each of the following sections describes part of how string text is encoded.

33.3.6.3.1 Standard Compression Scheme for Unicode (SCSU)

The Unicode consortium provides a standard text compression algorithm, which minimizes the amount of storage required for multiple-language strings. For more information, see “Links to UEFI-Related Documents” (http://uefi.org/uefi) under the heading “Unicode Compression Scheme”.

This specification extends the technique described in the following ways:

- The strings use the control code 0x7F to introduce the control codes described in Section 33.2.6.2.4. The following byte is the control code. The character value 0x7F will be encoded as 0x01 (SQ0) 0x7F.
- The language information contains default static and dynamic code windows, whereas SCSU provides fixed values for these.
- Characters between 0xF000 and 0xFCFF should be rejected.

33.3.6.3.2 Unicode 2-Byte Encoding (UCS-2)

The Unicode consortium provides a standard encoding algorithm, which takes two bytes per character. For more information see “Links to UEFI-Related Documents” (http://uefi.org/uefi) under the heading “Unicode Consortium”.

Characters between 0xF000 and 0xFCFF should be rejected.

33.3.7 Image Package

The Image package record describes the mapping between image identifiers and the pixels of the image themselves. The package consists of three parts: a fixed header, image information and the palette information.

33.3.7.1 Fixed Header

Summary

The fixed header consists of a standard record header and the offsets of the image and palette information.

Prototype

```c
typedef struct _EFI_HII_IMAGE_PACKAGE_HDR {
    EFI_HII_PACKAGE_HEADER Header;
    UINT32 ImageInfoOffset;
    UINT32 PaletteInfoOffset;
} EFI_HII_IMAGE_PACKAGE_HDR;
```

Members

- **Header**

  Standard package header, where Header.Type = EFI_HII_PACKAGE_IMAGES.
**ImageInfoOffset**
Offset, relative to this header, of the image information. If this is zero, then there are no images in the package.

**PaletteInfoOffset**
Offset, relative to this header, of the palette information. If this is zero, then there are no palettes in the image package.

### 33.3.7.2 Image Information
For each image identifier, the image information gives the bitmap and the relevant palette. The image information is encoded as a series of blocks, each with a single byte header. The blocks must be processed in order.

Each block begins with a single byte, which contains the block type.

```c
typedef struct _EFI_HII_IMAGE_BLOCK {
    UINT8 BlockType;
    UINT8 BlockBody[];
} EFI_HII_IMAGE_BLOCK;
```

![Image Information Encoded in Blocks](image.png)

Figure 33-43 Image Information Encoded in Blocks
The following table describes the different block types:

**Table 33-9 Block Types**

<table>
<thead>
<tr>
<th>Name</th>
<th>Value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>EFI_HII_IIBT_END</td>
<td>0x00</td>
<td>The end of the image information.</td>
</tr>
<tr>
<td>EFI_HII_IIBT_IMAGE_1BIT</td>
<td>0x10</td>
<td>1-bit w/palette</td>
</tr>
<tr>
<td>EFI_HII_IIBT_IMAGE_1BIT_TRANS</td>
<td>0x11</td>
<td>1-bit w/palette &amp; transparency</td>
</tr>
<tr>
<td>EFI_HII_IIBT_IMAGE_4BIT</td>
<td>0x12</td>
<td>4-bit w/palette</td>
</tr>
<tr>
<td>EFI_HII_IIBT_IMAGE_4BIT_TRANS</td>
<td>0x13</td>
<td>4-bit w/palette &amp; transparency</td>
</tr>
<tr>
<td>EFI_HII_IIBT_IMAGE_8BIT</td>
<td>0x14</td>
<td>8-bit w/palette</td>
</tr>
<tr>
<td>EFI_HII_IIBT_IMAGE_8BIT_TRANS</td>
<td>0x15</td>
<td>8-bit w/palette &amp; transparency</td>
</tr>
<tr>
<td>EFI_HII_IIBT_IMAGE_24BIT</td>
<td>0x16</td>
<td>24-bit RGB</td>
</tr>
<tr>
<td>EFI_HII_IIBT_IMAGE_24BIT_TRANS</td>
<td>0x17</td>
<td>24-bit RGB w/transparency</td>
</tr>
<tr>
<td>EFI_HII_IIBT_IMAGE_JPEG</td>
<td>0x18</td>
<td>JPEG encoded image</td>
</tr>
<tr>
<td>EFI_HII_IIBT_IMAGE_PNG</td>
<td>0x19</td>
<td>PNG encoded image</td>
</tr>
<tr>
<td>EFI_HII_IIBT_DUPLICATE</td>
<td>0x20</td>
<td>Duplicate an existing image identifier</td>
</tr>
<tr>
<td>EFI_HII_IIBT_SKIP2</td>
<td>0x21</td>
<td>Skip a certain number of image identifiers</td>
</tr>
<tr>
<td>EFI_HII_IIBT_SKIP1</td>
<td>0x22</td>
<td>Skip a certain number of image identifiers</td>
</tr>
<tr>
<td>EFI_HII_IIBT_EXT1</td>
<td>0x30</td>
<td>For future expansion (one byte length field)</td>
</tr>
<tr>
<td>EFI_HII_IIBT_EXT2</td>
<td>0x31</td>
<td>For future expansion (two byte length field)</td>
</tr>
<tr>
<td>EFI_HII_IIBT_EXT4</td>
<td>0x32</td>
<td>For future expansion (four byte length field)</td>
</tr>
</tbody>
</table>

In order to recreate all images, start at the first block and process them all until an **EFI_HII_IIBT_END_BLOCK** block is found. When processing the image blocks, each block refers to the current image identifier (ImageIdCurrent), which is initially set to one (1). Image blocks of an unknown type should be skipped. If they cannot be skipped, then processing halts.

### 33.3.7.2.1 EFI_HII_IIBT_END

**Summary**

Marks the end of the image information.

**Prototype**

```
#define EFI_HII_IIBT_END 0x00

typedef struct _EFI_HII_IIBT_END_BLOCK {
  EFI_HII_IMAGE_BLOCK Header;
} EFI_HII_IIBT_END_BLOCK;
```

**Members**

- **Header**

  Standard image block header, where Header.BlockType = **EFI_HII_IIBT_END_BLOCK**.
BlockType2

Indicates the type of extended block. See Section 33.3.6.2 for a list of all block types.

Description

Any images with an image identifier greater than or equal to ImageIdCurrent are empty.

33.3.7.2 EFI_HII_IIBT_EXT1, EFI_HII_IIBT_EXT2, EFI_HII_IIBT_EXT4

Summary

Generic prefix for image information with a 1-byte length.

Prototype

```c
#define EFI_HII_IIBT_EXT1 0x30
typedef struct _EFI_HII_IIBT_EXT1_BLOCK {
  EFI_HII_IMAGE_BLOCK  Header;
  UINT8    BlockType2;
  UINT8    Length;
} EFI_HII_IIBT_EXT1_BLOCK;

#define EFI_HII_IIBT_EXT2 0x31
typedef struct _EFI_HII_IIBT_EXT2_BLOCK {
  EFI_HII_IMAGE_BLOCK  Header;
  UINT8    BlockType2;
  UINT16   Length;
} EFI_HII_IIBT_EXT2_BLOCK;

#define EFI_HII_IIBT_EXT4 0x32
typedef struct _EFI_HII_IIBT_EXT4_BLOCK {
  EFI_HII_IMAGE_BLOCK  Header;
  UINT8    BlockType2;
  UINT32   Length;
} EFI_HII_IIBT_EXT4_BLOCK;
```

Members

- **Header**
  
  Standard image block header, where Header.BlockType = EFI_HII_IIBT_EXT1_BLOCK, EFI_HII_IIBT_EXT2_BLOCK or EFI_HII_IIBT_EXT4_BLOCK.

- **Length**
  
  Size of the image block, in bytes, including the image block header.

- **BlockType2**
  
  Indicates the type of extended block. See Section 33.3.7.2 for a list of all block types.
Description
Future extensions for image records which need a length-byte length use this prefix.

33.3.7.2.3 EFI_HII_IIBT_IMAGE_1BIT

Summary
One bit-per-pixel graphics image with palette information.

Prototype

```c
typedef struct _EFI_HII_IIBT_IMAGE_1BIT_BASE {
    UINT16 Width;
    UINT16 Height;
    UINT8 Data[ ... ];
} EFI_HII_IIBT_IMAGE_1BIT_BASE;

#define EFI_HII_IIBT_IMAGE_1BIT 0x10

typedef struct _EFI_HII_IIBT_IMAGE_1BIT_BLOCK {
    EFI_HII_IMAGE_BLOCK Header;
    UINT8 PaletteIndex;
    EFI_HII_IIBT_IMAGE_1BIT_BASE Bitmap;
} EFI_HII_IIBIT_IMAGE_1BIT_BLOCK;
```

Members

- **Header**
  Standard image header, where `Header.BlockType = EFI_HII_IIBT_IMAGE_1BIT`.

- **Width**
  Width of the bitmap in pixels.

- **Height**
  Height of the bitmap in pixels.

- **Bitmap**
  The bitmap specifies a series of pixels, one bit per pixel, left-to-right, top-to-bottom, and is padded out to the nearest byte. The number of bytes per bitmap can be calculated as: \((\text{Width} + 7)/8\) * \(\text{Height}\).

- **PaletteIndex**
  Index of the palette in the palette information.

Description
This record assigns the 1-bit-per-pixel bitmap data to the `ImageIdCurrent` identifier and increment `ImageIdCurrent` by one. The image’s upper left hand corner pixel is the most significant bit of the first bitmap byte. An example of a `EFI_HII_IIBT_IMAGE_1BIT` structure is shown below:
33.3.7.2.4 EFI_HII_IIBT_IMAGE_1BIT_TRANS

Summary

One bit-per-pixel graphics image with palette information and transparency.

Prototype

```c
#define EFI_HII_IIBT_IMAGE_1BIT_TRANS 0x11

typedef struct _EFI_HII_IIBT_IMAGE_1BIT_TRANS_BLOCK {
    EFI_HII_IMAGE_BLOCK Header;
    UINT8 PaletteIndex;
    EFI_HII_IIBT_IMAGE_1BIT_BASE Bitmap;
} EFI_HII_IIBT_IMAGE_1BIT_TRANS_BLOCK;
```

Members

*Header*

Standard image header, where `Header.BlockType = EFI_HII_IIBT_IMAGE_1BIT_TRANS`.

*PaletteIndex*

Index of the palette in the palette information.
**Bitmap**

The bitmap specifies a series of pixels, one bit per pixel, left-to-right, top-to-bottom, and is padded out to the nearest byte. The number of bytes per bitmap can be calculated as: \((\text{Width} + 7)/8 \times \text{Height}\).

**Description**

This record assigns the 1-bit-per-pixel bitmap data to the `ImageIdCurrent` identifier and increment `ImageIdCurrent` by one. The data in the `EFI_HII_IIBT_IMAGE_1BIT_TRANS` structure is exactly the same as the `EFI_HII_IIBT_IMAGE_1BIT` structure, the difference is how the data is treated.

The bitmap pixel value 0 is the ‘transparency’ value and will not be written to the screen. The bitmap pixel value 1 will be translated to the color specified by `Palette`.

### 33.3.7.2.5 EFI_HII_IIBT_IMAGE_24BIT

**Summary**

A 24 bit-per-pixel graphics image.

**Prototype**

```c
#define EFI_HII_IIBT_IMAGE_24BIT 0x16

typedef struct _EFI_HII_IIBT_IMAGE_24BIT_BASE
    UINT16 Width;
    UINT16 Height;
    EFI_HII_RGB_PIXEL Bitmap[ ... ];
} EFI_HII_IIBT_IMAGE_24BIT_BASE;

typedef struct _EFI_HII_IIBT_IMAGE_24BIT_BLOCK {
    EFI_HII_IMAGE_BLOCK Header;
    EFI_HII_IIBT_IMAGE_24BIT_BASE Bitmap;
} EFI_HII_IIBT_IMAGE_24BIT_BASE;
```

**Members**

- **Width**
  - Width of the bitmap in pixels.

- **Height**
  - Height of the bitmap in pixels.

- **Header**
  - Standard image header, where `Header.BlockType = EFI_HII_IIBT_IMAGE_24BIT`.

- **Bitmap**
  - The bitmap specifies a series of pixels, 24 bits per pixel, left-to-right, top-to-bottom. The number of bytes per bitmap can be calculated as: \((\text{Width} \times 3) \times \text{Height}\). Type `EFI_HII.RGB_PIXEL` is defined in “Related Definitions” below.
Description
This record assigns the 24-bit-per-pixel bitmap data to the `ImageIdCurrent` identifier and increment `ImageIdCurrent` by one. The image's upper left hand corner pixel is composed of the first three bitmap bytes. The first byte is the pixel's blue component value, the next byte is the pixel's green component value, and the third byte is the pixel's red component value (B,G,R). Each color component value can vary from 0x00 (color off) to 0xFF (color full on), allowing 16.8 millions colors that can be specified.

Related Definitions
```c
typedef struct _EFI_HII_RGB_PIXEL {
    UINT8 b;
    UINT8 g;
    UINT8 r;
} EFI_HII_RGB_PIXEL;
```

- `b` The relative intensity of blue in the pixel's color, from off (0x00) to full-on (0xFF).
- `g` The relative intensity of green in the pixel's color, from off (0x00) to full-on (0xFF).
- `r` The relative intensity of red in the pixel's color, from off (0x00) to full-on (0xFF).

33.3.7.2.6 EFI_HII_IIBT_IMAGE_24BIT_TRANS

Summary
A 24 bit-per-pixel graphics image with transparency.

Prototype
```c
#define _EFI_HII_IIBT_IMAGE_24BIT_TRANS 0x17

typedef struct EFI_HII_IIBT_IMAGE_24BIT_TRANS_BLOCK {
    EFI_HII_IMAGE_BLOCK Header;
    EFI_HII_IIBT_IMAGE_24BIT_BASE Bitmap;
} EFI_HII_IIBT_IMAGE_24BIT_TRANS_BLOCK;
```

Members
- `Header` Standard image header, where `Header.BlockType = EFI_HII_IIBT_IMAGE_24BIT_TRANS`.
- `Bitmap` The bitmap specifies a series of pixels, 24 bits per pixel, left-to-right, top-to-bottom. The number of bytes per bitmap can be calculated as: (Width * 3) * Height.
- `Width` Width of the bitmap in pixels.
**Height**

Height of the bitmap in pixels.

**Description**

This record assigns the 24-bit-per-pixel bitmap data to the `ImageIdCurrent` identifier and increment `ImageIdCurrent` by one. The data in the `EFI_HII_IMAGE_24BIT_TRANS` structure is exactly the same as the `EFI_HII_IMAGE_24BIT` structure, the difference is how the data is treated.

The bitmap pixel value 0x00, 0x00, 0x00 is the ‘transparency’ value and will not be written to the screen. All other bitmap pixel values will be written as defined to the screen. Since the ‘transparency’ value replaces true black, for image to display black they should use the color 0x00, 0x00, 0x01 (very dark red)

### 33.3.7.2.7 EFI_HII_IIBT_IMAGE_4BIT

**Summary**

Four bits-per-pixel graphics image with palette information.

**Prototype**

```c
typedef struct _EFI_HII_IIBT_IMAGE_4BIT_BASE {
    UINT16 Width;
    UINT16 Height;
    UINT8 Data[ ... ];
} EFI_HII_IIBT_IMAGE_4BIT_BASE;

#define EFI_HII_IIBT_IMAGE_4BIT 0x12

typedef struct _EFI_HII_IIBT_IMAGE_4BIT_BLOCK {
    EFI_HII_IMAGE_BLOCK Header;
    UINT8 PaletteIndex;
    EFI_HII_IIBT_IMAGE_4BIT_BASE Bitmap;
} EFI_HII_IIBT_IMAGE_4BIT_BLOCK;
```

**Members**

- **Width**
  Width of the bitmap in pixels.

- **Height**
  Height of the bitmap in pixels.

- **Header**
  Standard image header, where `Header.BlockType = EFI_HII_IIBT_IMAGE_4BIT`.

- **PaletteIndex**
  Index of the palette in the palette information.
**Bitmap**

The bitmap specifies a series of pixels, four bits per pixel, *left-to-right, top-to-bottom*, and is padded out to the nearest byte. The number of bytes per bitmap can be calculated as: \((\text{Width} + 1)/2\) * \(\text{Height}\).

**Description**

This record assigns the 4-bit-per-pixel bitmap data to the `ImageIdCurrent` identifier using the specified palette and increment `ImageIdCurrent` by one. The image’s upper left hand corner pixel is the most significant nibble of the first bitmap byte.

### 33.3.7.2.8 EFI_HII_IIBT_IMAGE_4BIT_TRANS

**Summary**

Four bits-per-pixel graphics image with palette information and transparency.

**Prototype**

```c
#define EFI_HII_IIBT_IMAGE_4BIT_TRANS 0x13
typedef struct _EFI_HII_IIBT_IMAGE_4BIT_TRANS_BLOCK {
    EFI_HII_IMAGE_BLOCK Header;
    UINT8 PaletteIndex;
    EFI_HII_IIBT_IMAGE_4BIT_BASE Bitmap;
} EFI_HII_IIBT_IMAGE_4BIT_TRANS_BLOCK;
```

**Members**

- **Header**
  Standard image header, where `Header.BlockType = EFI_HII_IIBT_IMAGE_4BIT_TRANS`.

- **PaletteIndex**
  Index of the palette in the palette information.

- **Bitmap**
  The bitmap specifies a series of pixels, four bits per pixel, *left-to-right, top-to-bottom*, and is padded out to the nearest byte. The number of bytes per bitmap can be calculated as: \((\text{Width} + 1)/2\) * \(\text{Height}\).

**Description**

This record assigns the 4-bit-per-pixel bitmap data to the `ImageIdCurrent` identifier using the specified palette and increment `ImageIdCurrent` by one. The data in the `EFI_HII_IMAGE_4BIT_TRANS` structure is exactly the same as the `EFI_HII_IMAGE_4BIT` structure, the difference is how the data is treated.

The bitmap pixel value 0 is the ‘transparency’ value and will not be written to the screen. All the other bitmap pixel values will be translated to the color specified by `Palette`. 
33.3.7.2.9 EFI_HII_IIBT_IMAGE_8BIT

Summary
Eight bits-per-pixel graphics image with palette information.

Prototype

```c
#define EFI_HII_IIBT_IMAGE_8BIT 0x14

typedef struct _EFI_HII_IIBT_IMAGE_8BIT_BASE {
    UINT16 Width;
    UINT16 Height;
    UINT8 Data[...];
} EFI_HII_IIBT_IMAGE_8BIT_BASE;

typedef struct _EFI_HII_IIBT_IMAGE_8BIT_BLOCK {
    EFI_HII_IMAGE_BLOCK Header;
    UINT8 PalettesIndex;
    EFI_HII_IIBT_IMAGE_8BIT_BASE Bitmap;
} EFI_HII_IIBT_IMAGE_8BIT_BLOCK;
```

Members

- **Width**
  Width of the bitmap in pixels.

- **Height**
  Height of the bitmap in pixels.

- **Header**
  Standard image header, where `Header.BlockType = EFI_HII_IIBT_IMAGE_8BIT`.

- **PalettesIndex**
  Index of the palette in the palette information.

- **Bitmap**
  The bitmap specifies a series of pixels, eight bits per pixel, left-to-right, top-to-bottom. The number of bytes per bitmap can be calculated as: `Width * Height`.

Description
This record assigns the 8-bit-per-pixel bitmap data to the `ImageIdCurrent` identifier using the specified palette and increment `ImageIdCurrent` by one. The image’s upper left hand corner pixel is the first bitmap byte.

33.3.7.2.10 EFI_HII_IIBT_IMAGE_8BIT_TRANS

Summary
Eight bits-per-pixel graphics image with palette information and transparency.
Prototype

```c
#define EFI_HII_IIBT_IMAGE_8BIT_TRANS 0x15

typedef struct _EFI_HII_IIBT_IMAGE_8BIT_TRANS_BLOCK {
    EFI_HII_IMAGE_BLOCK    Header;
    UINT8                  PaletteIndex;
    EFI_HII_IIBT_IMAGE_8BIT_BASE Bitmap;
} EFI_HII_IIBT_IMAGE_8BIT_TRANS_BLOCK;
```

Members

- **Header**
  Standard image header, where `Header.BlockType = EFI_HII_IIBT_IMAGE_8BIT_TRANS`.

- **PaletteIndex**
  Index of the palette in the palette information.

- **Bitmap**
  The bitmap specifies a series of pixels, eight bits per pixel, left-to-right, top-to-bottom. The number of bytes per bitmap can be calculated as: `Width * Height`.

Description

This record assigns the 8-bit-per-pixel bitmap data to the `ImageIdCurrent` identifier using the specified palette and increment `ImageIdCurrent` by one. The data in the `EFI_HII_IMAGE_8BIT_TRANS` structure is exactly the same as the `EFI_HII_IMAGE_8BIT` structure, the difference is how the data is treated.

The bitmap pixel value 0 is the ‘transparency’ value and will not be written to the screen. All the other bitmap pixel values will be translated to the color specified by `Palette`.

### 33.3.7.2.11 EFI_HII_IIBT_DUPLICATE

**Summary**

Assigns a new character value to a previously defined image.

Prototype

```c
#define EFI_HII_IIBT_DUPLICATE 0x20

typedef struct _EFI_HII_IIBT_DUPLICATE_BLOCK {
    EFI_HII_IMAGE_BLOCK    Header;
    EFI_IMAGE_ID           ImageId;
} EFI_HII_IIBT_DUPLICATE_BLOCK;
```

Members

- **Header**
  Standard image header, where `Header.BlockType = EFI_HII_IIBT_DUPLICATE`.
**ImageId**

The previously defined image ID with the exact same image.

**Description**

Indicates that the image with image ID `ImageValueCurrent` has the same image as a previously defined image ID and increments `ImageValueCurrent` by one.

### 33.3.7.2.12 EFI_HII_IIBT_IMAGE_JPEG

**Summary**

A true-color bitmap is encoded with JPEG image compression.

**Prototype**

```c
#define EFI_HII_IIBT_IMAGE_JPEG 0x18

typedef struct _EFI_HII_IIBT_JPEG_BLOCK {
  EFI_HII_IMAGE_BLOCK Header;
  UINT32 Size;
  UINT8 Data[...];
} EFI_HII_IIBT_JPEG;
```

**Members**

- **Header**
  Standard image header, where `Header.BlockType` = `EFI_HII_IIBT_IMAGE_JPEG`.

- **Size**
  Specifies the size of the JPEG encoded data.

- **Data**
  JPEG encoded data with ‘JFIF’ signature at offset 6 in the data block. The JPEG encoded data, specifies type of encoding and final size of true-color image.

**Description**

This record assigns the JPEG image data to the `ImageIdCurrent` identifier and increments `ImageIdCurrent` by one. The JPEG decoder is only required to cover the basic JPEG encoding types, which are produced by standard available paint packages (for example: MSPaint under Windows from Microsoft). This would include JPEG encoding of high (1:1:1) and medium (4:1:1) quality with only three components (R,G,B) – no support for the special gray component encoding.

### 33.3.7.2.13 EFI_HII_IIBT_SKIP1

**Summary**

Skips image IDs.
Prototype

#define EFI_HII_IIBT_SKIP1 0x22

typedef struct _EFI_HII_IIBT_SKIP1_BLOCK {
  EFI_HII_IMAGE_BLOCK Header;
  UINT8 SkipCount;
} EFI_HII_IIBT_SKIP1_BLOCK;

Members

  Header
  Standard image header, where Header.BlockType = EFI_HII_IIBT_SKIP1.

  SkipCount
  The unsigned 8-bit value to add to ImageIdCurrent.

Description

Increments the current image ID ImageIdCurrent by the number specified.

33.3.7.2.14 EFI_HII_IIBT_SKIP2

Summary

Skips image IDs.

Prototype

#define EFI_HII_IIBT_SKIP2 0x21

typedef struct _EFI_HII_IIBT_SKIP2_BLOCK {
  EFI_HII_IMAGE_BLOCK Header;
  UINT16 SkipCount;
} EFI_HII_IIBT_SKIP2_BLOCK;

Members

  Header
  Standard image header, where Header.BlockType = EFI_HII_IIBT_SKIP2.

  SkipCount
  The unsigned 16-bit value to add to ImageIdCurrent.

Description

Increments the current image ID ImageIdCurrent by the number specified.

33.3.7.2.15 EFI_HII_IIBT_PNG_BLOCK

Add a new image block structure for EFI_HII_IIBT_IMAGE_PNG. This supports the PNG image format in EFI HII image database.
Related Definitions

typedef struct _EFI_HII_IIBT_IMAGE_PNG {
    EFI_HII_IMAGE_BLOCK   Header;
    UINT32                Size;
    UINT8 Data[1];
} EFI_HII_IIBT_PNG_BLOCK;

Member

    Header   Standard image block header, where Header.locktype =
             EFI_HII_IIBT_IMAGE_PNG.
    Size     Size of the PNG image.
    Data     The raw data of the PNG image file.

33.3.7.3 Palette Information

Summary
This section describes the palette information within an image package.

Prototype

typedef struct _EFI_HII_IMAGE_PALETTE_INFO_HEADER {
    UINT16 PaletteCount;
} EFI_HII_IMAGE_PALETTE_INFO_HEADER;

Members

    PaletteCount
    Number of palettes.

Description
This fixed header is followed by zero or more variable-length palette information records. The structures
are assigned a number 1 to n.

33.3.7.3.1 Palette Information Records

Summary
A single palette
Prototype

typedef struct _EFI_HII_IMAGE_PALETTE_INFO {
  UINT16 PaletteSize;
  EFI_HII_RGB_PIXEL PaletteValue[...];
} EFI_HII_IMAGE_PALETTE_INFO;

Members

   PaletteSize
       Size of the palette information.

   PaletteValue
       Array of color values. Type EFI_HII_RGB_PIXEL is described in "Related Definitions" in EFI_HII_IIBT_IMAGE_24BIT.

Description

Each palette information record is an array of 24-bit color structures. The first entry (PaletteValue[0]) corresponds to color 0 in the source image; the second entry (PaletteValue[1]) corresponds to color 1, etc. Each palette entry is a three byte entry, with the first byte equal to the blue component of the color, followed by green, and finally red (B,G,R). Each color component value can vary from 0x00 (color off) to 0xFF (color full on), allowing 16.8 millions colors that can be specified.

A black & white 1-bit image would have the following palette structure:

\[
\text{PaletteSize} = 6
\]

\[
\begin{align*}
0. & \text{B(00) G(00) R(00) } \\
1. & \text{B(FF) G(FF) R(FF) } \\
\end{align*}
\]

Figure 33-44 Palette Structure of a Black & White, One-Bit Image
A 4-bit image would have the following palette structure:

![Palette Structure of a Four-Bit Image](image)

Figure 33-45 Palette Structure of a Four-Bit Image

The image palette must only contain the palette entries specified in the bitmap. The bitmap should allocate each color index starting from 0x00, so the palette information can be as small as possible. The following is an example of a palette structure of a 4-bit image that only uses 6 colors:
Each palette entry specifies each unique color in the image. The above figure would be typical of light blue logo on a black background, with several shades of blue for anti-aliasing the blue logo on the black background.

### 33.3.8 Forms Package

The Forms package is used to carry forms-based encoding data.

#### Prototype

```c
typedef struct _EFI_HII_FORM_PACKAGE_HDR {
    EFI_HII_PACKAGE_HEADER Header;
    //EFI_IFR_OP_HEADER OpCodeHeader;
    //More op-codes follow
} EFI_HII_FORM_PACKAGE_HDR;
```

#### Parameters

- **Header**
  
  The standard package header, where `Header.Type = EFI_HII_PACKAGE_FORMS`.

- **OpCodeHeader**
  
  The header for the first of what will be a series of op-codes associated with the forms data described in this package. The syntax of the forms can be referenced in [Section 33.2.5](#).  

#### Description

This is a package type designed to represent Internal Forms Representation (IFR) objects as a collection of op-codes.

### 33.3.8.1 Binary Encoding

The IFR is a binary encoding for HII-related objects. Every object has (at least) three attributes:
Opcode. The enumeration of all of the different HII-related objects.

Length. The length of the opcode itself (2-127 bytes).

Scope. If set, this opens up a new scope. Certain objects describe attributes or capabilities which only apply to the current scope rather than the entire form. The scope extends up to the special END opcode, which marks the end of the current scope.

The binary objects are encoded as byte stream. Every object begins with a standard header (EFI_IFR_OP_HEADER), which describes the opcode type, length and scope.

The simple binary object consists of a standard header, which contains a single 8-bit opcode, a 7-bit length and a 1-bit nesting indicator. The length specifies the number of bytes in the opcode, including the header. The simple binary object may also have zero or more bytes of fixed, object-specific, data.

---

**Figure 33-47 Simple Binary Object**

When the Scope bit is set, it marks the beginning of a new scope which applies to all subsequent opcodes until the matching EFI_IFR_END opcode is found to close the scope. Those opcodes may, in turn, open new scopes as well, creating nested scopes.

33.3.8.2 Standard Headers

33.3.8.2.1 EFI_IFR_OP_HEADER

**Summary**

Standard opcode header

**Prototype**

```c
typedef struct _EFI_IFR_OP_HEADER {
    UINT8 OpCode;
    UINT8 Length:7;
    UINT8 Scope:1;
} EFI_IFR_OP_HEADER;
```

**Members**

- **OpCode** Defines which type of operation is being described by this header. See Section 33.3.8.3 for a list of IFR opcodes.
- **Length** Defines the number of bytes in the opcode, including this header.
- **Scope** If this bit is set, the opcode begins a new scope, which is ended by an EFI_IFR_END opcode.

**Description**

Forms are represented in a binary format roughly similar to processor instructions.
Each header contains an opcode, a length and a scope indicator.

If Scope indicator is set, the scope exists until it reaches a corresponding EFI_IFR_END opcode. Scopes may be nested within other scopes.

Related Definitions

```c
typedef UINT16 EFI_QUESTION_ID;
typedef UINT16 EFI_IMAGE_ID;
typedef UINT16 EFI_STRING_ID;
typedef UINT16 EFI_FORM_ID;
typedef UINT16 EFI_VARSTORE_ID;
typedef UINT16 EFI_ANIMATION_ID;
```

### 33.3.8.2.2 EFI_IFR_QUESTION_HEADER

**Summary**

Standard question header.

**Prototype**

```c
typedef struct _EFI_IFR_QUESTION_HEADER {
  EFI_IFR_STATEMENT_HEADER Header;
  EFI_QUESTION_ID QuestionId;
  EFI_VARSTORE_ID VarStoreId;
  union {
    EFI_STRING_ID VarName;
    UINT16 VarOffset;
  } VarStoreInfo;
  UINT8 Flags;
} EFI_IFR_QUESTION_HEADER;
```

**Members**

- **Header**
  The standard statement header.

- **QuestionId**
  The unique value that identifies the particular question being defined by the opcode. The value of zero is reserved.

- **Flags**
  A bit-mask that determines which unique settings are active for this question. See “Related Definitions” below for the meanings of the individual bits.

- **VarStoreId**
  Specifies the identifier of a previously declared variable store to use when storing the question’s value. A value of zero indicates no associated variable store.

- **VarStoreInfo**
  If VarStoreId refers to Buffer Storage (EFI_IFR_VARSTORE or EFI_IFR_VARSTORE_EFI), then VarStoreInfo contains a 16-bit Buffer Storage offset (VarOffset). If VarStoreId refers to Name/Value Storage (EFI_IFR_VARSTORE_NAME_VALUE), then VarStoreInfo contains the String ID of the name (VarName) for this name/value pair.
Description
This is the standard header for questions.

Related Definitions

```c
// Flags values
#define EFI_IFR_FLAG_READ_ONLY           0x01
#define EFI_IFR_FLAG_CALLBACK            0x04
#define EFI_IFR_FLAG_RESET_REQUIRED      0x10
#define EFI_IFR_FLAG_REST_STYLE          0x20
#define EFI_IFR_FLAG_RECONNECT_REQUIRED  0x40
#define EFI_IFR_FLAG_OPTIONS_ONLY        0x80
```

EFI_IFR_FLAG_READ_ONLY The question is read-only

EFI_IFR_FLAG_CALLBACK Designates if a particular opcode is to be treated as something that will initiate a callback to a registered driver.

EFI_IFR_FLAG_RESET_REQUIRED If a particular choice is modified, designates that a return flag will be activated upon exiting of the browser, which indicates that the changes that the user requested require a reset to enact.

EFI_IFR_FLAG_REST_STYLE Designates if a question supports REST architectural style operation. This flag can be omitted if the formset class guid already contains EFI_HII_REST_STYLE_FORMSET_GUID.

EFI_IFR_FLAG_RECONNECT_REQUIRED If a particular choice is modified, designates that a return flag will be activated upon exiting of the formset or the browser, which indicates that the changes that the user requested require a reconnect to enact.

EFI_IFR_FLAG_OPTIONS_ONLY For questions with options, this indicates that only the options will be available for user choice.

### 33.3.8.2.3 EFI_IFR_STATEMENT_HEADER

Summary
Standard statement header.

Prototype

```c
typedef struct _EFI_IFR_STATEMENT_HEADER {
    EFI_STRING_ID Prompt;
    EFI_STRING_ID Help;
} EFI_IFR_STATEMENT_HEADER;
```
Members

- **Prompt**: The string identifier of the prompt string for this particular statement. The value 0 indicates no prompt string.
- **Help**: The string identifier of the help string for this particular statement. The value 0 indicates no help string.

Description

This is the standard header for statements, including questions.

33.3.8.3 Opcode Reference

This section describes each of the IFR opcode encodings in detail. The table below lists the opcodes in numeric order while the reference section lists them in alphabetic order.

<table>
<thead>
<tr>
<th>Opcode</th>
<th>Value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>EFI_IFR_FORM_OP</td>
<td>0x01</td>
<td>Form</td>
</tr>
<tr>
<td>EFI_IFR_SUBTITLE_OP</td>
<td>0x02</td>
<td>Subtitle statement</td>
</tr>
<tr>
<td>EFI_IFR_TEXT_OP</td>
<td>0x03</td>
<td>Static text/image statement</td>
</tr>
<tr>
<td>EFI_IFR_IMAGE_OP</td>
<td>0x04</td>
<td>Static image.</td>
</tr>
<tr>
<td>EFI_IFR_ONE_OF_OP</td>
<td>0x05</td>
<td>One-of question</td>
</tr>
<tr>
<td>EFI_IFR_CHECKBOX_OP</td>
<td>0x06</td>
<td>Boolean question</td>
</tr>
<tr>
<td>EFI_IFR_NUMERIC_OP</td>
<td>0x07</td>
<td>Numeric question</td>
</tr>
<tr>
<td>EFI_IFR_PASSWORD_OP</td>
<td>0x08</td>
<td>Password string question</td>
</tr>
<tr>
<td>EFI_IFR_ONE_OF_OPTION_OP</td>
<td>0x09</td>
<td>Option</td>
</tr>
<tr>
<td>EFI_IFR_SUPPRESS_IF_OP</td>
<td>0x0A</td>
<td>Suppress if conditional</td>
</tr>
<tr>
<td>EFI_IFR_LOCKED_OP</td>
<td>0x0B</td>
<td>Marks statement/question as locked</td>
</tr>
<tr>
<td>EFI_IFR_ACTION_OP</td>
<td>0x0C</td>
<td>Button question</td>
</tr>
<tr>
<td>EFI_IFR_RESET_BUTTON_OP</td>
<td>0x0D</td>
<td>Reset button statement</td>
</tr>
<tr>
<td>EFI_IFR_FORM_SET_OP</td>
<td>0x0E</td>
<td>Form set</td>
</tr>
<tr>
<td>EFI_IFR_REF_OP</td>
<td>0x0F</td>
<td>Cross-reference statement</td>
</tr>
<tr>
<td>EFI_IFR_NO_SUBMIT_IF_OP</td>
<td>0x10</td>
<td>Error checking conditional</td>
</tr>
<tr>
<td>EFI_IFR_INCONSISTENT_IF_OP</td>
<td>0x11</td>
<td>Error checking conditional</td>
</tr>
<tr>
<td>EFI_IFR_EQ_ID_VAL_OP</td>
<td>0x12</td>
<td>Return true if question value equals UINT16</td>
</tr>
<tr>
<td>EFI_IFR_EQ_ID_ID_OP</td>
<td>0x13</td>
<td>Return true if question value equals another question value</td>
</tr>
<tr>
<td>EFI_IFR_EQ_ID_VAL_LIST_OP</td>
<td>0x14</td>
<td>Return true if question value is found in list of UINT16s</td>
</tr>
<tr>
<td>EFI_IFR_AND_OP</td>
<td>0x15</td>
<td>Push true if both sub-expressions returns true.</td>
</tr>
<tr>
<td>Opcode</td>
<td>Value</td>
<td>Description</td>
</tr>
<tr>
<td>---------------------------</td>
<td>-------</td>
<td>----------------------------------------------------------------------------</td>
</tr>
<tr>
<td>EFI_IFR_OR_OP</td>
<td>0x16</td>
<td>Push true if either sub-expressions returns true.</td>
</tr>
<tr>
<td>EFI_IFR_NOT_OP</td>
<td>0x17</td>
<td>Push false if sub-expression returns true, otherwise return true.</td>
</tr>
<tr>
<td>EFI_IFR_RULE_OP</td>
<td>0x18</td>
<td>Create rule in current form.</td>
</tr>
<tr>
<td>EFI_IFR_GRAY_OUT_IF_OP</td>
<td>0x19</td>
<td>Nested statements, questions or options will not be selectable if expression returns true.</td>
</tr>
<tr>
<td>EFI_IFR_DATE_OP</td>
<td>0x1A</td>
<td>Date question.</td>
</tr>
<tr>
<td>EFI_IFR_TIME_OP</td>
<td>0x1B</td>
<td>Time question.</td>
</tr>
<tr>
<td>EFI_IFR_STRING_OP</td>
<td>0x1C</td>
<td>String question</td>
</tr>
<tr>
<td>EFI_IFR_REFRESH_OP</td>
<td>0x1D</td>
<td>Interval for refreshing a question</td>
</tr>
<tr>
<td>EFI_IFR_DISABLE_IF_OP</td>
<td>0x1E</td>
<td>Nested statements, questions or options will not be processed if expression returns true.</td>
</tr>
<tr>
<td>EFI_IFR_ANIMATION_OP</td>
<td>0x1F</td>
<td>Animation associated with question statement, form or form set.</td>
</tr>
<tr>
<td>EFI_IFR_TO_LOWER_OP</td>
<td>0x20</td>
<td>Convert a string on the expression stack to lower case.</td>
</tr>
<tr>
<td>EFI_IFR_TO_UPPER_OP</td>
<td>0x21</td>
<td>Convert a string on the expression stack to upper case.</td>
</tr>
<tr>
<td>EFI_IFR_MAP_OP</td>
<td>0x22</td>
<td>Convert one value to another by selecting a match from a list.</td>
</tr>
<tr>
<td>EFI_IFR_ORDERED_LIST_OP</td>
<td>0x23</td>
<td>Set question</td>
</tr>
<tr>
<td>EFI_IFR_VARSTORE_OP</td>
<td>0x24</td>
<td>Define a buffer-style variable storage.</td>
</tr>
<tr>
<td>EFI_IFR_VARSTORE_NAME_VALUE_OP</td>
<td>0x25</td>
<td>Define a name/value style variable storage.</td>
</tr>
<tr>
<td>EFI_IFR_VARSTORE_EFI_OP</td>
<td>0x26</td>
<td>Define a UEFI variable style variable storage.</td>
</tr>
<tr>
<td>EFI_IFR_VARSTORE_DEVICE_OP</td>
<td>0x27</td>
<td>Specify the device path to use for variable storage.</td>
</tr>
<tr>
<td>EFI_IFR_VERSION_OP</td>
<td>0x28</td>
<td>Push the revision level of the UEFI Specification to which this Forms Processor is compliant.</td>
</tr>
<tr>
<td>EFI_IFR_END_OP</td>
<td>0x29</td>
<td>Marks end of scope.</td>
</tr>
<tr>
<td>EFI_IFR_MATCH_OP</td>
<td>0x2A</td>
<td>Push TRUE if string matches a pattern.</td>
</tr>
<tr>
<td>EFI_IFR_GET_OP</td>
<td>0x2B</td>
<td>Return a stored value.</td>
</tr>
<tr>
<td>EFI_IFR_SET_OP</td>
<td>0x2C</td>
<td>Change a stored value.</td>
</tr>
<tr>
<td>EFI_IFR_READ_OP</td>
<td>0x2D</td>
<td>Provides a value for the current question or default.</td>
</tr>
<tr>
<td>EFI_IFR_WRITE</td>
<td>0x2E</td>
<td>Change a value for the current question.</td>
</tr>
<tr>
<td>EFI_IFR_EQUAL_OP</td>
<td>0x2F</td>
<td>Push TRUE if two expressions are equal.</td>
</tr>
<tr>
<td>EFI_IFR_NOT_EQUAL_OP</td>
<td>0x30</td>
<td>Push TRUE if two expressions are not equal.</td>
</tr>
<tr>
<td>EFI_IFR_GREATER_THAN_OP</td>
<td>0x31</td>
<td>Push TRUE if one expression is greater than another expression.</td>
</tr>
<tr>
<td>Opcode</td>
<td>Value</td>
<td>Description</td>
</tr>
<tr>
<td>-------------------------------</td>
<td>-------</td>
<td>-----------------------------------------------------------------------------</td>
</tr>
<tr>
<td>EFI_IFR_GREATER_EQUAL_OP</td>
<td>0x32</td>
<td>Push TRUE if one expression is greater than or equal to another expression.</td>
</tr>
<tr>
<td>EFI_IFR_LESS_THAN_OP</td>
<td>0x33</td>
<td>Push TRUE if one expression is less than another expression.</td>
</tr>
<tr>
<td>EFI_IFR_LESS_EQUAL_OP</td>
<td>0x34</td>
<td>Push TRUE if one expression is less than or equal to another expression.</td>
</tr>
<tr>
<td>EFI_IFR_BITWISE_AND_OP</td>
<td>0x35</td>
<td>Bitwise-AND two unsigned integers and push the result.</td>
</tr>
<tr>
<td>EFI_IFR_BITWISE_OR_OP</td>
<td>0x36</td>
<td>Bitwise-OR two unsigned integers and push the result.</td>
</tr>
<tr>
<td>EFI_IFR_BITWISE_NOT_OP</td>
<td>0x37</td>
<td>Bitwise-NOT an unsigned integer and push the result.</td>
</tr>
<tr>
<td>EFI_IFR_SHIFT_LEFT_OP</td>
<td>0x38</td>
<td>Shift an unsigned integer left by a number of bits and push the result.</td>
</tr>
<tr>
<td>EFI_IFR_SHIFT_RIGHT_OP</td>
<td>0x39</td>
<td>Shift an unsigned integer right by a number of bits and push the result.</td>
</tr>
<tr>
<td>EFI_IFR_ADD_OP</td>
<td>0x3A</td>
<td>Add two unsigned integers and push the result.</td>
</tr>
<tr>
<td>EFI_IFR_SUBTRACT_OP</td>
<td>0x3B</td>
<td>Subtract two unsigned integers and push the result.</td>
</tr>
<tr>
<td>EFI_IFR_MULTIPLY_OP</td>
<td>0x3C</td>
<td>Multiply two unsigned integers and push the result.</td>
</tr>
<tr>
<td>EFI_IFR_DIVIDE_OP</td>
<td>0x3D</td>
<td>Divide one unsigned integer by another and push the result.</td>
</tr>
<tr>
<td>EFI_IFR_MODULO_OP</td>
<td>0x3E</td>
<td>Divide one unsigned integer by another and push the remainder.</td>
</tr>
<tr>
<td>EFI_IFR_RULE_REF_OP</td>
<td>0x3F</td>
<td>Evaluate a rule</td>
</tr>
<tr>
<td>EFI_IFR_QUESTION_REF1_OP</td>
<td>0x40</td>
<td>Push a question’s value</td>
</tr>
<tr>
<td>EFI_IFR_QUESTION_REF2_OP</td>
<td>0x41</td>
<td>Push a question’s value</td>
</tr>
<tr>
<td>EFI_IFR_UINT8_OP</td>
<td>0x42</td>
<td>Push an 8-bit unsigned integer</td>
</tr>
<tr>
<td>EFI_IFR_UINT16_OP</td>
<td>0x43</td>
<td>Push a 16-bit unsigned integer</td>
</tr>
<tr>
<td>EFI_IFR_UINT32_OP</td>
<td>0x44</td>
<td>Push a 32-bit unsigned integer</td>
</tr>
<tr>
<td>EFI_IFR_UINT64_OP</td>
<td>0x45</td>
<td>Push a 64-bit unsigned integer</td>
</tr>
<tr>
<td>EFI_IFR_TRUE_OP</td>
<td>0x46</td>
<td>Push a boolean TRUE</td>
</tr>
<tr>
<td>EFI_IFR_FALSE_OP</td>
<td>0x47</td>
<td>Push a boolean FALSE</td>
</tr>
<tr>
<td>EFI_IFR_TO_UINT8_OP</td>
<td>0x48</td>
<td>Convert expression to an unsigned integer</td>
</tr>
<tr>
<td>EFI_IFR_TO_STRING_OP</td>
<td>0x49</td>
<td>Convert expression to a string</td>
</tr>
<tr>
<td>EFI_IFR_TO_BOOLEAN_OP</td>
<td>0x4A</td>
<td>Convert expression to a boolean</td>
</tr>
<tr>
<td>EFI_IFR_MID_OP</td>
<td>0x4B</td>
<td>Extract portion of string or buffer</td>
</tr>
<tr>
<td>EFI_IFR_FIND_OP</td>
<td>0x4C</td>
<td>Find a string in a string</td>
</tr>
<tr>
<td>EFI_IFR_TOKEN_OP</td>
<td>0x4D</td>
<td>Extract a delimited byte or character string from buffer or string.</td>
</tr>
</tbody>
</table>
### Code Definitions

Each of the following sections gives a detailed description of the opcodes’ behavior.

#### 33.3.8.3.1 EFI_IFR_ACTION

**Summary**

Create an action button.

<table>
<thead>
<tr>
<th>Opcode</th>
<th>Value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>EFI_IFR_STRING_REF1_OP</td>
<td>0x4E</td>
<td>Push a string</td>
</tr>
<tr>
<td>EFI_IFR_STRING_REF2_OP</td>
<td>0x4F</td>
<td>Push a string</td>
</tr>
<tr>
<td>EFI_IFR_CONDITIONAL_OP</td>
<td>0x50</td>
<td>Duplicate one of two expressions depending on result of the first expression.</td>
</tr>
<tr>
<td>EFI_IFR_QUESTION_REF3_OP</td>
<td>0x51</td>
<td>Push a question’s value from a different form.</td>
</tr>
<tr>
<td>EFI_IFR_ZERO_OP</td>
<td>0x52</td>
<td>Push a zero</td>
</tr>
<tr>
<td>EFI_IFR_ONE_OP</td>
<td>0x53</td>
<td>Push a one</td>
</tr>
<tr>
<td>EFI_IFR_ONES_OP</td>
<td>0x54</td>
<td>Push a 0xFFFFFFFFFFFFFFFF.</td>
</tr>
<tr>
<td>EFI_IFR_UNDEFINED_OP</td>
<td>0x55</td>
<td>Push Undefined</td>
</tr>
<tr>
<td>EFI_IFR_LENGTH_OP</td>
<td>0x56</td>
<td>Push length of buffer or string.</td>
</tr>
<tr>
<td>EFI_IFR_DUP_OP</td>
<td>0x57</td>
<td>Duplicate top of expression stack</td>
</tr>
<tr>
<td>EFI_IFR_THIS_OP</td>
<td>0x58</td>
<td>Push the current question’s value</td>
</tr>
<tr>
<td>EFI_IFR_SPAN_OP</td>
<td>0x59</td>
<td>Return first matching/non-matching character in a string</td>
</tr>
<tr>
<td>EFI_IFR_VALUE_OP</td>
<td>0x5A</td>
<td>Provide a value for a question</td>
</tr>
<tr>
<td>EFI_IFR_DEFAULT_OP</td>
<td>0x5B</td>
<td>Provide a default value for a question.</td>
</tr>
<tr>
<td>EFI_IFR_DEFAULTSTORE_OP</td>
<td>0x5C</td>
<td>Define a Default Type Declaration</td>
</tr>
<tr>
<td>EFI_IFR_FORM_MAP_OP</td>
<td>0x5D</td>
<td>Create a standards-map form.</td>
</tr>
<tr>
<td>EFI_IFR_CATENATE_OP</td>
<td>0x5E</td>
<td>Push concatenated buffers or strings.</td>
</tr>
<tr>
<td>EFI_IFR_GUID_OP</td>
<td>0x5F</td>
<td>An extensible GUIDed op-code</td>
</tr>
<tr>
<td>EFI_IFR_SECURITY_OP</td>
<td>0x60</td>
<td>Returns whether current user profile contains specified setup access privileges.</td>
</tr>
<tr>
<td>EFI_IFR_MODAL_TAG_OP</td>
<td>0x61</td>
<td>Specify current form is modal</td>
</tr>
<tr>
<td>EFI_IFR_REFRESH_ID_OP</td>
<td>0x62</td>
<td>Establish an event group for refreshing a forms-based element.</td>
</tr>
<tr>
<td>EFI_IFR_WARNING_IF</td>
<td>0x63</td>
<td>Warning conditional</td>
</tr>
<tr>
<td>EFI_IFR_MATCH2_OP</td>
<td>0x64</td>
<td>Push TRUE if string matches a Regular Expression pattern.</td>
</tr>
</tbody>
</table>
Prototype

```
#define EFI_IFR_ACTION_OP 0x0C
typedef struct _EFI_IFR_ACTION {
    EFI_IFR_OP_HEADER Header;
    EFI_IFR_QUESTION_HEADER Question;
    EFI_STRING_ID QuestionConfig;
} EFI_IFR_ACTION;
```

```
#define EFI_IFR_ACTION_1 {
    EFI_IFR_OP_HEADER Header;
    EFI_IFR_QUESTION_HEADER Question;
} EFI_IFR_ACTION;
```

Members

- **Header**: The standard opcode header, where `Header.OpCode = EFI_IFR_ACTION_OP`
- **Question**: The standard question header. See `EFI_IFR_QUESTION_HEADER` (Section 33.3.8.2.2) for more information.
- **QuestionConfig**: The results string which is in `<ConfigResp>` format will be processed when the button is selected by the user.

Description

Creates an action question. When the question is selected, the configuration string specified by `QuestionConfig` will be processed. If `QuestionConfig` is 0 or is not present, then no configuration string will be processed. This is useful when using an action button only for the callback.

If the question is marked read-only (see `EFI_IFR_QUESTION_HEADER`) then the action question cannot be selected.

33.3.8.3.2 EFI_IFR_ANIMATION

Summary

Creates an image for a statement or question.

Prototype

```
#define EFI_IFR_ANIMATION_OP 0xF
typedef struct _EFI_IFR_ANIMATION {
    EFI_IFR_OP_HEADER Header;
    EFI_ANIMATION_ID Id;
} EFI_IFR_ANIMATION;
```

Members

- **Header**: Standard opcode header, where `Header.OpCode` is `EFI_IFR_ANIMATION_OP`
- **Id**: Animation identifier in the HII database.
Description
Associates an animation from the HII database with the current question, statement or form. If the specified animation does not exist in the HII database.

33.3.8.3.3 EFI_IFR_ADD

Summary
Pops two unsigned integers, adds them and pushes the result.

Prototype
```
#define EFI_IFR_ADD_OP 0x3a
typedef struct _EFI_IFR_ADD {
  EFI_IFR_OP_HEADER Header;
} EFI_IFR_ADD;
```

Members

Description
This opcode performs the following actions:

1. Pop two values from the expression stack. The first popped is the **right-hand** value. The second popped is the **left-hand** value.
2. If the two values do not evaluate to unsigned integers, push Undefined.
3. Zero-extend the **left-hand** and **right-hand** values to 64-bits.
4. Add the left-hand value to right-hand value.
5. Push the lower 64-bits of the result. Overflow is ignored.

33.3.8.3.4 EFI_IFR_AND

Summary
Pops two booleans, push **TRUE** if both are **TRUE**, otherwise push **FALSE**.

Prototype
```
#define EFI_IFR_AND_OP 0x15
typedef struct _EFI_IFR_AND {
  EFI_IFR_OP_HEADER Header;
} EFI_IFR_AND;
```

Members
- **Header**: The standard opcode header, where `Header.OpCode = EFI_IFR_AND_OP`.

Description
This opcode performs the following actions:
1. Pop two expressions from the expression stack.
2. If the two expressions cannot be evaluated as boolean, push Undefined.
3. If both expressions evaluate to **TRUE**, then push **TRUE**. Otherwise, push **FALSE**.

### 33.3.8.3.5 EFI_IFR_BITWISE_AND

**Summary**

Pops two unsigned integers, perform bitwise AND and push the result.

**Prototype**

```c
#define EFI_IFR_BITWISE_AND_OP 0x35
typedef struct _EFI_IFR_BITWISE_AND {
  EFI_IFR_OP_HEADER Header;
} EFI_IFR_BITWISE_AND;
```

**Members**

- **Header**
  

**Description**

This opcode performs the following actions:

1. Pop two expressions from the expression stack.
2. If the two expressions cannot be evaluated as unsigned integers, push Undefined.
3. Otherwise, zero-extend the unsigned integers to 64-bits.
4. Perform a bitwise-AND on the two values.
5. Push the result.

### 33.3.8.3.6 EFI_IFR_BITWISE_NOT

**Summary**

Pop an unsigned integer, perform a bitwise NOT and push the result.

**Prototype**

```c
#define EFI_IFR_BITWISE_NOT_OP 0x37
typedef struct _EFI_IFR_BITWISE_NOT {
  EFI_IFR_OP_HEADER Header;
} EFI_IFR_BITWISE_NOT;
```

**Members**

- **Header**
  
  The standard opcode header, where `Header.OpCode = EFI_IFR_BITWISE_NOT_OP`.

**Description**

This opcode performs the following actions:

1. Pop an expression from the expression stack.
2. If the expression cannot be evaluated as an unsigned integer, push Undefined.
3. Otherwise, zero-extend the unsigned integer to 64-bits.
4. Perform a bitwise-NOT on the value.
5. Push the result.

### 33.3.8.3.7 EFI_IFR_BITWISE_OR

**Summary**

Pops two unsigned integers, perform bitwise OR and push the result.

**Prototype**

```c
#define EFI_IFR_BITWISE_OR_OP 0x36
typedef struct _EFI_IFR_BITWISE_OR {
    EFI_IFR_OP_HEADER Header;
} EFI_IFR_BITWISE_OR;
```

**Members**

- **Header**
  
  Standard opcode header, where **OpCode** is **EFI_IFR_BITWISE_OR_OP**.

**Description**

This opcode performs the following actions:
1. Pop two expressions from the expression stack.
2. If the two expressions cannot be evaluated as unsigned integers, push Undefined.
3. Otherwise, zero-extend the unsigned integers to 64-bits.
4. Perform a bitwise-OR of the two values.
5. Push the result.

### 33.3.8.3.8 EFI_IFR_CATENATE

**Summary**

Pops two buffers or strings, concatenates them and pushes the result.

**Prototype**

```c
#define EFI_IFR_CATENATE_OP 0x5e
typedef struct _EFI_IFR_CATENATE {
    EFI_IFR_OP_HEADER Header;
} EFI_IFR_CATENATE;
```

**Members**

- **Header**
  
  Standard opcode header, where **OpCode** is **EFI_IFR_CATENATE_OP**.

**Description**

This opcode performs the following actions:
1. Pop two expressions from the expression stack. The first expression popped is the left value and the second value popped is the right value.

2. If the left or right values cannot be evaluated as a string or a buffer, push Undefined. If the left or right values are of different types, then push Undefined.

3. If the left and right values are strings, push a new string which contains the contents of the left string (without the NULL terminator) followed by the contents of the right string on to the expression stack.

4. If the left and right values are buffers, push a new buffer that contains the contents of the left buffer followed by the contents of the right buffer on to the expression stack.

33.3.8.3.9 EFI_IFR_CHECKBOX

Summary

Creates a boolean checkbox.

Prototype

```c
#define EFI_IFR_CHECKBOX_OP 0x06
typedef struct _EFI_IFR_CHECKBOX {
    EFI_IFR_OP_HEADER   Header;
    EFI_IFR_QUESTION_HEADER Question;
    UINT8                Flags;
} EFI_IFR_CHECKBOX;
```

Members

- **Header**
  - The standard question header, where `Header.OpCode = EFI_IFR_CHECKBOX_OP`.

- **Question**
  - The standard question header. See `EFI_IFR_QUESTION_HEADER` (Section 33.3.8.2.2) for more information.

- **Flags**
  - Flags that describe the behavior of the question. All undefined bits should be zero. See `EFI_IFR_CHECKBOX_x` in "Related Definitions" for more information.

Description

Creates a Boolean checkbox question and adds it to the current form. The checkbox has two values: `FALSE` if the box is not checked and `TRUE` if it is.

There are three ways to specify defaults for this question: the Flags field (lowest priority), one or more nested `EFI_IFR_ONE_OF_OPTION`, or nested `EFI_IFR_DEFAULT` (highest priority).

An image may be associated with the question using a nested `EFI_IFR_IMAGE`. An animation may be associated with the option using a nested `EFI_IFR_ANIMATION`. 
Related Definitions

#define EFI_IFR_CHECKBOX_DEFAULT 0x01
#define EFI_IFR_CHECKBOX_DEFAULT_MFG 0x02

33.3.8.3.10 EFI_IFR_CONDITIONAL

Summary
Pops two values and a boolean, pushes one of the values depending on the boolean.

Prototype

#define EFI_IFR_CONDITIONAL_OP 0x50
typedef struct _EFI_IFR_CONDITIONAL {
    EFI_IFR_OP_HEADER Header;
} EFI_IFR_CONDITIONAL;

Members

Header Standard opcode header, where OpCode is EFI_IFR_CONDITIONAL_OP.

Description
This opcode performs the following actions:

1. Pop three values from the expression stack. The first value popped is the right value. The second expression popped is the middle value. The last expression popped is the left value.
2. If the left value cannot be evaluated as a boolean, push Undefined.
3. If the left expression evaluates to TRUE, push the right value.
4. Otherwise, push the middle value.

33.3.8.3.11 EFI_IFR_DATE

Summary
Create a date question.

Prototype

#define EFI_IFR_DATE_OP 0x1A
typedef struct _EFI_IFR_DATE {
    EFI_IFR_OP_HEADER Header;
    EFI_IFR_QUESTION_HEADER Question;
    UINT8 Flags;
} EFI_IFR_DATE;

Members

Header The sequence that defines the type of opcode as well as the length of the opcode being defined. Header.OpCode = EFI_IFR_DATE_OP.
**Question**
The standard question header. See [Section 33.3.8.2.2](#) for more information.

**Flags**
Flags that describe the behavior of the question. All undefined bits should be zero.

```c
#define EFI_QF_DATE_YEAR_SUPPRESS 0x01
#define EFI_QF_DATE_MONTH_SUPPRESS 0x02
#define EFI_QF_DATE_DAY_SUPPRESS 0x04
#define EFI_QF_DATE_STORAGE 0x30
```

For **QF_DATE_STORAGE**, there are currently three valid values:

```c
#define QF_DATE_STORAGE_NORMAL   0x00
#define QF_DATE_STORAGE_TIME     0x10
#define QF_DATE_STORAGE_WAKEUP   0x20
```

**Description**
Create a Date question (see [Section 33.2.5.4.6](#)) and add it to the current form.

There are two ways to specify defaults for this question: one or more nested **EFI_IFR_ONE_OF_OPTION** (lowest priority) or nested **EFI_IFR_DEFAULT** (highest priority). An image may be associated with the option using a nested **EFI_IFR_IMAGE**. An animation may be associated with the question using a nested **EFI_IFR_ANIMATION**.

### 33.3.8.3.12 EFI_IFR_DEFAULT

**Summary**
Provides a default value for the current question

**Prototype**

```c
#define EFI_IFR_DEFAULT_OP 0x5b
typedef struct _EFI_IFR_DEFAULT {
  EFI_IFR_OP_HEADER Header;
  UINT16             DefaultId;
  UINT8              Type;
  EFI_IFR_TYPE_VALUE Value;
} EFI_IFR_DEFAULT;

typedef struct _EFI_IFR_DEFAULT_2 {
  EFI_IFR_OP_HEADER Header;
  UINT16             DefaultId;
  UINT8              Type;
} EFI_IFR_DEFAULT_2;
```
Members

Header
The sequence that defines the type of opcode as well as the length of the opcode being defined. For this tag, Header.OpCode = EFI_IFR_DEFAULT_OP.

DefaultId
Identifies the default store for this value. The default store must have previously been created using EFI_IFR_DEFAULTSTORE.

Type
The type of data in the Value field. See EFI_IFR_TYPE_x in EFI_IFR_ONE_OF_OPTION.

Value
The default value. The actual size of this field depends on Type. If Type is EFI_IFR_TYPE_OTHER, then the default value is provided by a nested EFI_IFR_VALUE.

Description
This opcode specifies a default value for the current question. There are two forms. The first (EFI_IFR_DEFAULT) assumes that the default value is a constant, embedded directly in the Value member. The second (EFI_IFR_DEFAULT_2) assumes that the default value is specified using a nested EFI_IFR_VALUE opcode.

33.3.8.3.13 EFI_IFR_DEFAULTSTORE

Summary
Provides a declaration for the type of default values that a question can be associated with.

Prototype

```c
#define EFI_IFR_DEFAULTSTORE_OP 0x5c
typedef struct _EFI_IFR_DEFAULTSTORE {
    EFI_IFR_OP_HEADER Header;
    EFI_STRING_ID DefaultName;
    UINT16 DefaultId;
} EFI_IFR_DEFAULTSTORE;
```

Members

Header
The sequence that defines the type of opcode as well as the length of the opcode being defined. For this tag, Header.OpCode = EFI_IFR_DEFAULTSTORE_OP.

DefaultName
A string token reference for the human readable string associated with the type of default being declared.

DefaultId
The default identifier, which is unique within the current form set. The default identifier creates a group of defaults. See Section for the default identifier ranges.

Description
Declares a class of default which can then have question default values associated with.

An EFI_IFR_DEFAULTSTORE with a specified DefaultId must appear in the IFR before it can be referenced by an EFI_IFR_DEFAULT.
33.3.8.3.14 EFI_IFR_DISABLE_IF

Summary
Disable all nested questions and expressions if the expression evaluates to **TRUE**.

Prototype
```
#define EFI_IFR_DISABLE_IF_OP 0x1e
typedef struct _EFI_IFR_DISABLE_IF {
  EFI_IFR_OP_HEADER Header;
} EFI_IFR_DISABLE_IF;
```

Members
- **Header**
  The byte sequence that defines the type of opcode as well as the length of the opcode being defined. `Header.OpCode` is `EFI_IFR_DISABLE_IF_OP`.

Description
All nested statements, questions, options or expressions will not be processed if the expression appearing as the first nested object evaluates to TRUE. If the expression consists of more than a single opcode, then the first opcode in the expression must have the Scope bit set and the expression must end with `EFI_IFR_END`.

When this opcode appears under a form set, the expression must only rely on constants. When this opcode appears under a form, the expression may rely on question values in the same form which are not inside of an `EFI_DISABLE_IF` expression.

33.3.8.3.15 EFI_IFR_DIVIDE

Summary
Pops two unsigned integers, divide one by the other and pushes the result.

Prototype
```
#define EFI_IFR_DIVIDE_OP 0x3d
typedef struct _EFI_IFR_DIVIDE {
  EFI_IFR_OP_HEADER Header;
} EFI_IFR_DIVIDE;
```

Members
- **Header**
  Standard opcode header, where OpCode is `EFI_IFR_DIVIDE`.

Description
1. Pop two expressions from the expression stack. The first popped is the right-hand expression. The second popped is the left-hand expression.
2. If the two expressions do not evaluate to unsigned integers, push Undefined. If the right-hand expression is equal to zero, push Undefined.
3. Zero-extend the left-hand and right-hand expressions to 64-bits.
4. Divide the left-hand value to right-hand expression.
5. Push the result.

33.3.8.3.16 EFI_IFR_DUP

Summary
Duplicate the top value on the expression stack.

Prototype
```
#define EFI_IFR_DUP_OP 0x57
typedef struct _EFI_IFR_DUP {
    EFI_IFR_OP_HEADER   Header;
} EFI_IFR_DUP;
```

Members
- Header: Standard opcode header, where OpCode is EFI_IFR_DUP_OP.

Description
Duplicate the top expression on the expression stack.

Note: This opcode is usually used as an optimization by the tools to help eliminate common sub-expression calculation and make smaller expressions.

33.3.8.3.17 EFI_IFR_END

Summary
End of the current scope.

Prototype
```
#define EFI_IFR_END_OP 0x29
typedef struct _EFI_IFR_END {
    EFI_IFR_OP_HEADER   Header;
} EFI_IFR_END;
```

Members
- Header: Standard opcode header, where OpCode is EFI_IFR_END_OP.

Description
Marks the end of the current scope.

33.3.8.3.18 EFI_IFR_EQUAL

Summary
Pop two values, compare and push TRUE if equal, FALSE if not.
Prototype

```c
#define EFI_IFR_EQUAL_OP 0x2f
typedef struct _EFI_IFR_EQUAL {
    EFI_IFR_OP_HEADER Header;
} EFI_IFR_EQUAL;
```

Members

- **Header**: Standard opcode header, where **OpCode** is **EFI_IFR_EQUAL_OP**.

Description

The opcode performs the following actions:

1. Pop two values from the expression stack.
2. If the two values are not strings, Booleans or unsigned integers, push Undefined.
3. If the two values are of different types, push Undefined.
4. Compare the two values. Strings are compared lexicographically.
5. If the two values are equal then push **TRUE** on the expression stack. If they are not equal, push **FALSE**.

33.3.8.3.19 EFI_IFR_EQ_ID_ID

Summary

Push **TRUE** if the two questions have the same value or **FALSE** if they are not equal.

Prototype

```c
#define EFI_IFR_EQ_ID_ID_OP 0x13
typedef struct _EFI_IFR_EQ_ID_ID {
    EFI_IFR_OP_HEADER Header;
    EFI_QUESTION_ID QuestionId1;
    EFI_QUESTION_ID QuestionId2;
} EFI_IFR_EQ_ID_ID;
```

Members

- **Header**: Standard opcode header, where **OpCode** is **EFI_IFR_EQ_ID_ID_OP**.
- **QuestionId1, QuestionId2**: Specifies the identifier of the questions whose values will be compared.

Description

Evaluate the values of the specified questions (**QuestionId1, QuestionId2**). If the two values cannot be evaluated or cannot be converted to comparable types, then push Undefined. If they are equal, push **TRUE**. Otherwise push **FALSE**.
33.3.8.3.20 EFI_IFR_EQ_ID_VAL_LIST

**Summary**
Push **TRUE** if the question’s value appears in a list of unsigned integers.

**Prototype**

```c
#define EFI_IFR_EQ_ID_VAL_LIST_OP 0x14
typedef struct _EFI_IFR_EQ_ID_VAL_LIST {
    EFI_IFR_OP_HEADER Header;
    EFI_QUESTION_ID QuestionId;
    UINT16 ListLength;
    UINT16 ValueList[1];
} EFI_IFR_EQ_ID_VAL_LIST;
```

**Members**
- **Header**
  - Standard opcode header, where OpCode is `EFI_IFR_EQ_ID_VAL_LIST_OP`.
- **QuestionId**
  - Specifies the identifier of the question whose value will be compared.
- **ListLength**
  - Number of entries in `ValueList`.
- **ValueList**
  - Zero or more unsigned integer values to compare against.

**Description**
Evaluate the value of the specified question (`QuestionId`). If the specified question cannot be evaluated as an unsigned integer, then push Undefined. If the value can be found in `ValueList`, then push **TRUE**. Otherwise push **FALSE**.

33.3.8.3.21 EFI_IFR_EQ_ID_VAL

**Summary**
Push **TRUE** if a question’s value is equal to a 16-bit unsigned integer, otherwise **FALSE**.

**Prototype**

```c
#define EFI_IFR_EQ_ID_VAL_OP 0x12
typedef struct _EFI_IFR_EQ_ID_VAL {
    EFI_IFR_OP_HEADER Header;
    EFI_QUESTION_ID QuestionId;
    UINT16 Value;
} EFI_IFR_EQ_ID_VAL;
```

**Members**
- **Header**
  - Standard opcode header, where OpCode is `EFI_IFR_EQ_ID_VAL_OP`.
- **QuestionId**
  - Specifies the identifier of the question whose value will be compared.
- **Value**
  - Unsigned integer value to compare against.
### Description
Evaluate the value of the specified question (QuestionId). If the specified question cannot be evaluated as an unsigned integer, then push Undefined. If they are equal, push TRUE. Otherwise push FALSE.

#### 33.3.8.3.22 EFI_IFR_FALSE

**Summary**
Push a FALSE on to the expression stack.

**Prototype**
```c
#define EFI_IFR_FALSE_OP 0x47
typedef struct _EFI_IFR_FALSE {
    EFI_IFR_OP_HEADER Header;
} EFI_IFR_FALSE;
```

**Members**
- **Header**: The sequence that defines the type of opcode as well as the length of the opcode being defined. For this tag, Header.OpCode = EFI_IFR_FALSE_OP

**Description**
Push a FALSE on to the expression stack.

#### 33.3.8.3.23 EFI_IFR_FIND

**Summary**
Pop two strings and an unsigned integer, find one string in the other and the index where found.

**Prototype**
```c
#define EFI_IFR_FIND_OP 0x4c
typedef struct _EFI_IFR_FIND {
    EFI_IFR_OP_HEADER Header;
    UINT8 Format;
} EFI_IFR_FIND;
```

**Members**
- **Header**: Standard opcode header, where OpCode is EFI_IFR_FIND_OP.
- **Format**: The following flags govern the matching criteria:

**Related Definitions**
- `#define EFI_IFR_FF_CASE_SENSITIVE   0x00`
- `#define EFI_IFR_FF_CASE_INSENSITIVE  0x01`

**Description**
This opcode performs the following actions:
1. Pop three expressions from the expression stack. The first expression popped is the right-hand value and the second value popped is the middle value and the last value popped is the left-hand value.

2. If the left-hand or middle values cannot be evaluated as a string, push Undefined. If the third expression cannot be evaluated as an unsigned integer, push Undefined.

3. The left-hand value is the string to search. The middle value is the string to compare with. The right-hand expression is the zero-based index of the search.

4. If the string is found, push the zero-based index of the found string.

5. Otherwise, if the string is not found or the right-hand value specifies a value which is greater-than or equal to the length of the left-hand value’s string, push 0xFFFFFFFFFFFFFFFF.

33.3.8.3.24 EFI_IFR_FORM

Summary
Creates a form.

Prototype

```c
#define EFI_IFR_FORM_OP 0x01
typedef struct _EFI_IFR_FORM {
    EFI_IFR_OP_HEADER Header;
    EFI_FORM_ID FormId;
    EFI_STRING_ID FormTitle;
} EFI_IFR_FORM;
```

Members

- **Header**: The sequence that defines the type of opcode as well as the length of the opcode being defined. `Header.OpCode = EFI_IFR_FORM_OP`.

- **FormId**: The form identifier, which uniquely identifies the form within the form set. The form identifier, along with the device path and form set GUID, uniquely identifies a form within a system.

- **FormTitle**: The string token reference to the title of this particular form.

Description

A form is the encapsulation of what amounts to a browser page. The header defines a FormId, which is referenced by the form set, among others. It also defines a FormTitle, which is a string to be used as the title for the form.

33.3.8.3.25 EFI_IFR_FORM_MAP

Summary

Creates a standards map form.
Prototype

```c
#define EFI_IFR_FORM_MAP_OP 0x5D
typedef struct _EFI_IFR_FORM_MAP_METHOD {
    EFI_STRING_ID       MethodTitle;
    EFI_GUID            MethodIdentifier;
} EFI_IFR_FORM_MAP_METHOD;

typedef struct _EFI_IFR_FORM_MAP {
    EFI_IFR_OP_HEADER   Header;
    EFI_FORM_ID         FormId;
    //EFI_IFR_FORM_MAP_METHOD   Methods[];
} EFI_IFR_FORM_MAP;
```

Parameters

- **Header**
  The sequence that defines the type of opcode as well as the length of the opcode being defined. `Header.OpCode = EFI_IFR_FORM_MAP_OP`.
- **FormId**
  The unique identifier for this particular form.
- **Methods**
  One or more configuration method’s name and unique identifier.
- **MethodTitle**
  The string identifier which provides the human-readable name of the configuration method for this standards map form.
- **MethodIdentifier**
  Identifier which uniquely specifies the configuration methods associated with this standards map form. See “Related Definitions” for current identifiers.

Description

A standards map form describes how the configuration settings are represented for a configuration method identified by `MethodIdentifier`. It also defines a `FormTitle`, which is a string to be used as the title for the form.

Related Definitions

```c
#define EFI_HII_STANDARD_FORM_GUID \
    { 0x3bd2f4ec, 0xe524, 0x46e4, \
      { 0xa9, 0xd8, 0x51, 0x01, 0x17, 0x42, 0x55, 0x62 } }
```

An `EFI_IFR_FORM_MAP` where the method identifier is `EFI_HII_STANDARD_FORM_GUID` is semantically identical to a normal `EFI_IFR_FORM`.

33.3.8.3.26 EFI_IFR_FORM_SET

Summary

The form set is a collection of forms that are intended to describe the pages that will be displayed to the user.
Prototype

```c
#define EFI_IFR_FORM_SET_OP 0x0E

typedef struct _EFI_IFR_FORM_SET {
    EFI_IFR_OP_HEADER   Header;
    EFI_GUID              Guid;
    EFI_STRING_ID        FormSetTitle;
    EFI_STRING_ID        Help;
    UINT8                Flags;
    //EFI_GUID        ClassGuid[...];
} EFI_IFR_FORM_SET;
```

Members

- **Header**: The sequence that defines the type of opcode as well as the length of the opcode being defined. `Header.OpCode = EFI_IFR_FORM_SET_OP`.
- **Guid**: The unique GUID value associated with this particular form set. Type ` EFI_GUID` is defined in `InstallProtocolInterface()` in this specification.
- **FormSetTitle**: The string token reference to the title of this particular form set.
- **Help**: The string token reference to the help of this particular form set.
- **Flags**: Flags which describe additional features of the form set. Bits 0:1 = number of members in `ClassGuid`. Bits 2:7 = Reserved. Should be set to zero.
- **ClassGuid**: Zero to four class identifiers. The standard class identifiers are described in `EFI_HII_FORM_BROWSER2_PROTOCOL.SendForm()`. They do not need to be unique in the form set.

Description

The form set consists of a header and zero or more forms.

### 33.3.8.3.27 EFI_IFR_GET

**Summary**

Return a stored value.
Prototype

```c
#define EFI_IFR_GET_OP 0x2B
typedef struct _EFI_IFR_GET {
    EFI_IFR_OP_HEADER Header;
    EFI_VARSTORE_ID VarStoreId;
    union {
        EFI_STRING_ID VarName;
        UINT16 VarOffset;
    } VarStoreInfo;
    UINT8 VarStoreType;
} EFI_IFR_GET;
```

Parameters

- **Header**: The sequence that defines the type of opcode as well as the length of the opcode being defined. `Header.OpCode = EFI_IFR_GET_OP`.
- **VarStoreId**: Specifies the identifier of a previously declared variable store to use when retrieving the value.
- **VarStoreInfo**: Depending on the type of variable store selected, this contains either a 16-bit Buffer Storage offset (`VarOffset`) or a Name/Value or EFI Variable name (`VarName`).
- **VarStoreType**: Specifies the type used for storage. The storage types `EFI_IFR_TYPE_x` are defined in `EFI_IFR_ONE_OF_OPTION`.

Description

This operator takes the value from storage and pushes it on to the expression stack. If the value could not be retrieved from storage, then `Undefined` is pushed on to the expression stack.

The type of value retrieved from storage depends on the setting of `VarStoreType`, as described in the following table:

<table>
<thead>
<tr>
<th>VarStoreType</th>
<th>Storage Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>EFI_IFR_TYPE_NUM_SIZE_8</code></td>
<td>8-bit unsigned integer</td>
</tr>
<tr>
<td><code>EFI_IFR_TYPE_NUM_SIZE_16</code></td>
<td>16-bit unsigned integer</td>
</tr>
<tr>
<td><code>EFI_IFR_TYPE_NUM_SIZE_32</code></td>
<td>32-bit unsigned integer</td>
</tr>
<tr>
<td><code>EFI_IFR_TYPE_NUM_SIZE_64</code></td>
<td>64-bit unsigned integer</td>
</tr>
<tr>
<td><code>EFI_IFR_TYPE_BOOLEAN</code></td>
<td>8-bit boolean (0 = false, 1 = true)</td>
</tr>
<tr>
<td><code>EFI_IFR_TYPE_TIME</code></td>
<td><code>EFI_HII_TIME</code></td>
</tr>
<tr>
<td><code>EFI_IFR_TYPE_DATE</code></td>
<td><code>EFI_HII_DATE</code></td>
</tr>
</tbody>
</table>
33.3.8.3.28 EFI_IFR_GRAY_OUT_IF

Summary
Creates a group of statements or questions which are conditionally grayed-out.

Prototype
#define EFI_IFR_GRAY_OUT_IF_OP 0x19
typedef struct _EFI_IFR_GRAY_OUT_IF {
   EFI_IFR_OP_HEADER Header;
} EFI_IFR_GRAY_OUT_IF;

Members
Header The byte sequence that defines the type of opcode as well as the length of the opcode being defined. Header.OpCode = EFI_IFR_GRAY_OUT_IF_OP.

Description
All nested statements or questions will be grayed out (not selectable and visually distinct) if the expression appearing as the first nested object evaluates to TRUE. If the expression consists of more than a single opcode, then the first opcode in the expression must have the Scope bit set and the expression must end with EFI_IFR_END.

Different browsers may support this option to varying degrees. For example, HTML has no similar construct so it may not support this facility.

33.3.8.3.29 EFI_IFR_GREATER_EQUAL

Summary
Pop two values, compare, push TRUE if first is greater than or equal the second, otherwise push FALSE.
Prototype

\[
\text{#define EFI_IFR_GREATER_EQUAL_OP 0x32}\\
\text{typedef struct _EFI_IFR_GREATER_EQUAL} \{\\
\text{ EFI_IFR_OP_HEADER Header;}\\
\} \text{ EFI_IFR_GREATER_EQUAL;}\\
\]

Members

\begin{itemize}
  \item \textbf{Header} \quad \text{Standard opcode header, where OpCode is EFI_IFR_GREATER_EQUAL_OP.}
\end{itemize}

Description

This opcode performs the following actions:

1. Pop two values from the expression stack. The first value popped is the right-hand value and the second value popped is the left-hand value.
2. If the two values do not evaluate to string, boolean or unsigned integer, push Undefined.
3. If the two values do not evaluate to the same type, push Undefined.
4. Compare the two values. Strings are compared lexicographically.
5. If the left-hand value is greater than or equal to the right-hand value, push \textbf{TRUE}. Otherwise push \textbf{FALSE}.

33.3.8.3.30 EFI_IFR_GREATER_THAN

Summary

Pop two values, compare, push \textbf{TRUE} if first is greater than the second, otherwise push \textbf{FALSE}.

Prototype

\[
\text{#define EFI_IFR_GREATER_THAN_OP 0x31}\\
\text{typedef struct _EFI_IFR_GREATER_THAN} \{\\
\text{ EFI_IFR_OP_HEADER Header;}\\
\} \text{ EFI_IFR_GREATER_THAN;}\\
\]

Members

\begin{itemize}
  \item \textbf{Header} \quad \text{Standard opcode header, where OpCode is EFI_IFR_GREATER_THAN_OP}
\end{itemize}

Description

This opcode performs the following actions:

1. Pop two values from the expression stack. The first value popped is the right-hand value and the second value popped is the left-hand value.
2. If the two values do not evaluate to string, boolean or unsigned integer, push Undefined.
3. If the two values do not evaluate to the same type, push Undefined.
4. Compare the two values. Strings are compared lexicographically.
5. If the left-hand value is greater than the right-hand value, push \textbf{TRUE}. Otherwise push \textbf{FALSE}.\]
33.3.8.3.31 EFI_IFR_GUID

Summary
A GUIDed operation. This op-code serves as an extensible op-code which can be defined by the Guid value to have various functionality. It should be noted that IFR browsers or scripts which cannot interpret the meaning of this GUIDed op-code will skip it.

Prototype
```c
#define EFI_IFR_GUID_OP 0xF
typedef struct _EFI_IFR_GUID {
    EFI_IFR_OP_HEADER       Header;
    EFI_GUID                Guid;
    //Optional Data Follows
} EFI_IFR_GUID;
```

Parameters
- **Header**: The sequence that defines the type of opcode as well as the length of the opcode being defined. For this tag, `Header.OpCode = EFI_IFR_GUID_OP`
- **Guid**: The GUID value for this op-code. This field is intended to define a particular type of special-purpose function, and the format of the data which immediately follows the Guid field (if any) is defined by that particular GUID.

33.3.8.3.32 EFI_IFR_IMAGE

Summary
Creates an image for a statement or question.

Prototype
```c
#define EFI_IFR_IMAGE_OP 0x4
typedef struct _EFI_IFR_IMAGE {
    EFI_IMAGE_ID        Id;
} EFI_IFR_IMAGE;
```

Members
- **Id**: Image identifier in the HII database.

Description
Specifies the image within the HII database.

33.3.8.3.33 EFI_IFR_INCONSISTENT_IF

Summary
Creates a validation expression and error message for a question.
Prototype

```c
#define EFI_IFR_INCONSISTENT_IF_OP 0x011
typedef struct _EFI_IFR_INCONSISTENT_IF {
    EFI_IFR_OP_HEADER Header;
    EFI_STRING_ID Error;
} EFI_IFR_INCONSISTENT_IF;
```

Members

- **Header**: The byte sequence that defines the type of opcode as well as the length of the opcode being defined. `Header.OpCode = EFI_IFR_INCONSISTENT_IF_OP`.
- **Error**: The string token reference to the string that will be used for the consistency check message.

Description

This tag uses a Boolean expression to allow the IFR creator to check options in a richer manner than provided by the question tags themselves. For example, this tag might be used to validate that two options are not using the same address or that the numbers that were entered align to some pattern (such as leap years and February in a date input field). The tag provides a string to be used in an error display to alert the user to the issue. Inconsistency tags will be evaluated when the user traverses from tag to tag. The user should not be allowed to submit the results of a form inconsistency.

33.3.8.3.34 EFI_IFR_LENGTH

Summary

Pop a string or buffer, push its length.

Prototype

```c
#define EFI_IFR_LENGTH_OP 0x56
typedef struct _EFI_IFR_LENGTH {
    EFI_IFR_OP_HEADER Header;
} EFI_IFR_LENGTH;
```

Members

- **Header**: Standard opcode header, where `OpCode` is `EFI_IFR_LENGTH_OP`.

Description

This opcode performs the following actions:

1. Pop a value from the expression stack.
2. If the value cannot be evaluated as a buffer or string, then push Undefined.
3. If the value can be evaluated as a buffer, push the length of the buffer, in bytes.
4. If the value can be evaluated as a string, push the length of the string, in characters.
33.3.8.3.35 EFI_IFR_LESS_EQUAL

Summary
Pop two values, compare, push TRUE if first is less than or equal to the second, otherwise push FALSE.

Prototype
```
#define EFI_IFR_LESS_EQUAL_OP 0x34
typedef struct _EFI_IFR_LESS_EQUAL {
  EFI_IFR_OP_HEADER Header;
} EFI_IFR_LESS_EQUAL;
```

Members
Header Standard opcode header, where OpCode is EFI_IFR_LESS_EQUAL_OP.

Description
This opcode performs the following actions:
1. Pop two values from the expression stack. The first value popped is the right-hand value and the second value popped is the left-hand value.
2. If the two values do not evaluate to string, boolean or unsigned integer, push Undefined.
3. If the two values do not evaluate to the same type, push Undefined.
4. Compare the two values. Strings are compared lexicographically.
5. If the left-hand value is less than or equal to the right-hand value, push TRUE. Otherwise push FALSE.

33.3.8.3.36 EFI_IFR_LESS_THAN

Summary
Pop two values, compare, push TRUE if the first is less than the second, otherwise push FALSE.

Prototype
```
#define EFI_IFR_LESS_THAN_OP 0x33
typedef struct _EFI_IFR_LESS_THAN {
  EFI_IFR_OP_HEADER Header;
} EFI_IFR_LESS_THAN;
```

Members
Header Standard opcode header, where OpCode is EFI_IFR_LESS_THAN_OP.

Description
This opcode performs the following actions:
1. Pop two values from the expression stack. The first value popped is the right-hand value and the second value popped is the left-hand value.
2. If the two values do not evaluate to string, boolean or unsigned integer, push Undefined.
3. If the two values do not evaluate to the same type, push Undefined.
4. Compare the two values. Strings are compared lexicographically.
5. If the left-hand value is less than the right-hand value, push TRUE. Otherwise push FALSE.

### 33.3.8.3.37 EFI_IFR_LOCKED

**Summary**
Specifies that the statement or question is locked.

**Prototype**

```c
#define EFI_IFR_LOCKED_OP 0x0B
typedef struct _EFI_IFR_LOCKED {
    EFI_IFR_OP_HEADER Header;
} EFI_IFR_LOCKED;
```

**Parameters**
- **Header**
  Standard opcode header, where `Header.Opcode` is `EFI_IFR_LOCKED_OP`.

**Members**
None

**Description**
The presence of `EFI_IFR_LOCKED` indicates that the statement or question should not be modified by a Forms Editor.

### 33.3.8.3.38 EFI_IFR_MAP

**Summary**
Pops value, compares against an array of comparison values, pushes the corresponding result value.

**Prototype**

```c
#define EFI_IFR_MAP_OP 0x22
typedef struct _EFI_IFR_MAP {
    EFI_IFR_OP_HEADER Header;
} EFI_IFR_MAP;
```

**Parameters**
- **Header**
The sequence that defines the type of opcode as well as the length of the opcode being defined. For this tag, `Header.Opcode` = `EFI_IFR_MAP_OP`

**Description**
This operator contains zero or more expression pairs nested within its scope. Each expression pair contains a match expression and a return expression.
This opcode performs the following actions:

1. This operator pops a single value from the expression stack.
2. Compare this value against the evaluated result of each of the match expressions.
3. If there is a match, then the evaluated result of the corresponding return expression is pushed on to the expression stack.
4. If there is no match, then Undefined is pushed.

33.3.8.3.39 EFI_IFR_MATCH

Summary
Pop a source string and a pattern string, push **TRUE** if the source string matches the pattern specified by the pattern string, otherwise push **FALSE**.

Prototype

```c
#define EFI_IFR_MATCH_OP 0x2a
typedef struct _EFI_IFR_MATCH {
    EFI_IFR_OP_HEADER Header;
} EFI_IFR_MATCH;
```

Members

- **Header** Standard opcode header, where `Header.OpCode` is `EFI_IFR_MATCH_OP`.

Description

1. Pop two values from the expression stack. The first value popped is the string and the second value popped is the pattern.
2. If the string or the pattern cannot be evaluated as a string, then push Undefined.
3. Process the string and pattern using the `MetaiMatch` function of the `EFI_UNICODE_COLLATION2_PROTOCOL`.
4. If the result is **TRUE**, then push **TRUE**.
5. If the result is **FALSE**, then push **FALSE**.

33.3.8.3.40 EFI_IFR_MID

Summary
Pop a string or buffer and two unsigned integers, push an extracted portion of the string or buffer.

Prototype

```c
#define EFI_IFR_MID_OP 0x4b
typedef struct _EFI_IFR_MID {
    EFI_IFR_OP_HEADER Header;
} EFI_IFR_MID;
```

Members

- **Header** Standard opcode header, where ` OpCode` is `EFI_IFR_MID_OP`. 
Description

1. Pop three values from the expression stack. The first value popped is the right value and the second value popped is the middle value and the last expression popped is the left value.

2. If the left value cannot be evaluated as a string or a buffer, push Undefined. If the middle or right value cannot be evaluated as unsigned integers, push Undefined.

3. If the left value is a string, then the middle value is the 0-based index of the first character in the string to extract and the right value is the length of the string to extract. If the right value is zero or the middle value is greater than or equal the string’s length, then push an Empty string. Push the extracted string on the expression stack. If the right value would cause extraction to extend beyond the end of the string, then only the characters up to and include the last character of the string are in the pushed result.

4. If the left value is a buffer, then the middle value is the 0-based index of the first byte in the buffer to extract and the right value is the length of the buffer to extract. If the right value is zero or the middle value is greater than the buffer’s length, then push an empty buffer. Push the extracted buffer on the expression stack. If the right value would cause extraction to extend beyond the end of the buffer, then only the bytes up to and include the last byte of the buffer are in the pushed result.

33.3.8.3.41 EFI_IFR_MODAL_TAG

Summary

Specify that the current form is a modal form.

Prototype

```c
#define EFI_IFR_MODAL_TAG_OP 0x61
typedef struct _EFI_IFR_MODAL_TAG {
    EFI_IFR_OP_HEADER    Header;
} EFI_IFR_MODAL_TAG;
```

Members

- **Header**
  - Standard opcode header, where OpCode is `EFI_IFR_MODAL_TAG_OP`.

Description

When this opcode is present within the scope of a form, the form is modal; if the opcode is not present, the form is not modal.

A “modal” form is one that requires specific user interaction before it is deactivated. Examples of modal forms include error messages or confirmation dialogs.

When a modal form is activated, it is also selected. A modal form is deactivated only when one of the following occurs:

- The user chooses a “Navigate To Form” behavior (defined in Section 33.2.10.1.2, “Selected Form”). Note that this is distinct from the “Navigate Forms” behavior.
- A question in the form requires callback, and the callback returns one of the following ActionRequest values (defined in `EFI_HII_CONFIG_ACCESS_PROTOCOL.CallBack()`):
  - `EFI_BROWSER_ACTION_REQUEST_RESET`
A modal form cannot be deactivated using other navigation behaviors, including:

- Navigate Forms
- Exit Browser/Discard All (except when initiated by a callback as indicated above)
- Exit Browser/Submit All (except when initiated by a callback as indicated above)
- Exit Browser/Discard All/Reset Platform (except when initiated by a callback as indicated above)

### 33.3.8.3.42 EFI_IFR_MODULO

**Summary**

Pop two unsigned integers, divide one by the other and push the remainder.

**Prototype**

```c
define EFI_IFR_MODULO_OP 0x3e
typedef struct EFI_IFR_MODULO {
    EFI_IFR_OP_HEADER Header;
} EFI_IFR_MODULO;
```

**Members**

- **Header** Standard opcode header, where OpCode is EFI_IFR_MODULO_OP.

**Description**

This opcode performs the following actions:

1. Pop two values from the expression stack. The first value popped is the right-hand value and the second value popped is the left-hand value.
2. If the two values do not evaluate to unsigned integers, push Undefined. If the right-hand value to 0, push Undefined.
3. Zero-extend the values to 64-bits. Then, divide the left-hand value by the right-hand value.
4. Push the difference between the left-hand value and the product of the right-hand value and the calculated quotient.

### 33.3.8.3.43 EFI_IFR_MULTIPLY

**Summary**

Multiply one unsigned integer by another and push the result.
Prototype

```c
#define EFI_IFR_MULTIPLY_OP 0x3c
typedef struct _EFI_IFR_MULTIPLY {
    EFI_IFR_OP_HEADER Header;
} EFI_IFR_MULTIPLY;
```

Members

- **Header**: Standard opcode header, where `OpCode` is `EFI_IFR_MULTIPLY_OP`.

Description

This opcode performs the following actions:

1. Pop two values from the expression stack. The first value popped is the right-hand expression and the second value popped is the left-hand expression.
2. If the two values do not evaluate to unsigned integers, push Undefined.
3. Zero-extend the values to 64-bits. Then, multiply the right-hand value by the left-hand value. Push the lower 64-bits of the result.

### 33.3.8.3.44 EFI_IFR_NOT

Summary

Pop a boolean and, if `TRUE`, push `FALSE`. If `FALSE`, push `TRUE`.

Prototype

```c
#define EFI_IFR_NOT_OP 0x17
typedef struct _EFI_IFR_NOT {
    EFI_IFR_OP_HEADER Header;
} EFI_IFR_NOT;
```

Members

- **Header**: The sequence that defines the type of opcode as well as the length of the opcode being defined. `Header.OpCode` is `EFI_IFR_NOT_OP`.

Description

This opcode performs the following actions:

1. Pop one value from the expression stack.
2. If the value cannot be evaluated as a Boolean, push Undefined.
3. If the value evaluates to `TRUE`, then push `FALSE`. Otherwise, push `TRUE`.

### 33.3.8.3.45 EFI_IFR_NOT_EQUAL

Summary

Pop two values, compare and push `TRUE` if not equal, otherwise push `FALSE`. 
Prototype

```c
#define EFI_IFR_NOT_EQUAL_OP 0x30
typedef struct __EFI_IFR_NOT_EQUAL {
    EFI_IFR_OP_HEADER Header;
} EFI_IFR_NOT_EQUAL;
```

Members

- **Header**: Standard opcode header, where `OpCode` is `EFI_IFR_NOT_EQUAL_OP`.

Description

This opcode performs the following actions:

1. Pop two values from the expression stack.
2. If the two values are not strings, Booleans or unsigned integers, push Undefined.
3. If the two values are of different types, push Undefined.
4. Compare the two values. Strings are compared lexicographically.
5. If the two values are not equal then push **TRUE** on the expression stack. If they are equal, push **FALSE**.

### 33.3.8.3.46 EFI_IFR_NO_SUBMIT_IF

Summary

Creates a validation expression and error message for a question.

Prototype

```c
#define EFI_IFR_NO_SUBMIT_IF_OP 0x10
typedef struct __EFI_IFR_NO_SUBMIT_IF {
    EFI_IFR_OP_HEADER Header;
    EFI_STRING_ID Error;
} EFI_IFR_NO_SUBMIT_IF;
```

Members

- **Header**: The byte sequence that defines the type of opcode as well as the length of the opcode being defined. `Header.OpCode = EFI_IFR_NO_SUBMIT_IF_OP`.
- **Error**: The string token reference to the string that will be used for the consistency check message.

Description

Creates a conditional expression which will be evaluated when the form is submitted. If the conditional evaluates to **TRUE**, then the error message `Error` will be displayed to the user and the user will be prevented from submitting the form.
**33.3.8.3.47 EFI_IFR_NUMERIC**

**Summary**

Creates a number question.

**Prototype**

```c
#define EFI_IFR_NUMERIC_OP 0x07
typedef struct _EFI_IFR_NUMERIC {
    EFI_IFR_OP_HEADER Header;
    EFI_IFR_QUESTION_HEADER Question;
    UINT8 Flags;

    union {
        struct {
            UINT8 MinValue;
            UINT8 MaxValue;
            UINT8 Step;
        } u8;
        struct {
            UINT16 MinValue;
            UINT16 MaxValue;
            UINT16 Step;
        } u16;
        struct {
            UINT32 MinValue;
            UINT32 MaxValue;
            UINT32 Step;
        } u32;
        struct {
            UINT64 MinValue;
            UINT64 MaxValue;
            UINT64 Step;
        } u64;
    } data;
} EFI_IFR_NUMERIC;
```

**Members**

- **Header**
  The sequence that defines the type of opcode as well as the length of the opcode being defined. `Header.OpCode = EFI_IFR_NUMERIC_OP`.

- **Question**
  The standard question header. See Section 33.3.8.2.2 for more information.

- **Flags**
  Specifies flags related to the numeric question. See “Related Definitions”
MinValue

The minimum value to be accepted by the browser for this opcode. The size of the data field may vary from 8 to 64 bits.

MaxValue

The maximum value to be accepted by the browser for this opcode. The size of the data field may vary from 8 to 64 bits.

Step

Defines the amount to increment or decrement the value each time a user requests a value change. If the step value is 0, then the input mechanism for the numeric value is to be free-form and require the user to type in the actual value. The size of the data field may vary from 8 to 64 bits.

Description

Creates a number question on the current form, with built-in error checking and default information. The storage size depends on the EFI_IFR_NUMERIC_SIZE portion of the Flags field.

There are two ways to specify defaults for this question: one or more nested EFI_IFR_ONE_OF_OPTION (lowest priority) or nested EFI_IFR_DEFAULT (highest priority). An image may be associated with the option using a nested EFI_IFR_IMAGE. An animation may be associated with the question using a nested EFI_IFR_ANIMATION.

Related Definitions

```c
#define EFI_IFR_NUMERIC_SIZE     0x03
#define  EFI_IFR_NUMERIC_SIZE_1   0x00
#define  EFI_IFR_NUMERIC_SIZE_2   0x01
#define  EFI_IFR_NUMERIC_SIZE_4   0x02
#define  EFI_IFR_NUMERIC_SIZE_8   0x03

#define EFI_IFR_DISPLAY        0x30
#define  EFI_IFR_DISPLAY_INT_DEC   0x00
#define  EFI_IFR_DISPLAY_UINT_DEC  0x10
#define  EFI_IFR_DISPLAY_UINT_HEX  0x20
```

EFI_IFR_NUMERIC_SIZE

Specifies the size of the numeric value, the storage required and the size of the MinValue, MaxValue and Step values in the opcode header.

EFI_IFR_DISPLAY

The value will be displayed in signed decimal, unsigned decimal or unsigned hexadecimal. Input is still allowed in any form.

Note: IFR expressions do not support signed types (see Section 33.2.5.7.4 Data Types). The value of a numeric question is treated during expression evaluation as an unsigned integer even if EFI_IFR_DISPLAY_INT_DEC flag is specified. However, the EFI_IFR_DISPLAY_INT_DEC flag is taken into consideration while validating question’s current or default value against MinValue and MaxValue. When EFI_IFR_DISPLAY_INT_DEC flag is specified, forms
processor must treat MinValue, MaxValue, current question value, and default question value as signed integers.

33.3.8.3.48 EFI_IFR_ONE

Summary
Push a one on to the expression stack.

Prototype
```c
#define EFI_IFR_ONE_OP 0x53
typedef struct _EFI_IFR_ONE {
   EFI_IFR_OP_HEADER  Header;
} EFI_IFR_ONE;
```

Members
- **Header**
  The sequence that defines the type of opcode as well as the length of the opcode being defined. For this tag, `Header.OpCode = EFI_IFR_ONE_OP`

Description
Push a one on to the expression stack.

33.3.8.3.49 EFI_IFR_ONES

Summary
Push 0xFFFFFFFFFFFFFFFF on to the expression stack.

Prototype
```c
#define EFI_IFR_ONES_OP 0x54
typedef struct _EFI_IFR_ONES {
   EFI_IFR_OP_HEADER  Header;
} EFI_IFR_ONES;
```

Members
- **Header**
  The sequence that defines the type of opcode as well as the length of the opcode being defined. For this tag, `Header.OpCode = EFI_IFR_ONES_OP`

Description
Push 0xFFFFFFFFFFFFFFFF on to the expression stack.

33.3.8.3.50 EFI_IFR_ONE_OF

Summary
Creates a select-one-of question.
Prototype

```c
#define EFI_IFR_ONE_OF_OP 0x05

typedef struct _EFI_IFR_ONE_OF {
    EFI_IFR_OP_HEADER     Header;
    EFI_IFR_QUESTION_HEADER Question;
    UINT8                Flags;

    union {
        struct {
            UINT8     MinValue;
            UINT8     MaxValue;
            UINT8     Step;
        } u8;
        struct {
            UINT16    MinValue;
            UINT16    MaxValue;
            UINT16    Step;
        } u16;
        struct {
            UINT32    MinValue;
            UINT32    MaxValue;
            UINT32    Step;
        } u32;
        struct {
            UINT64    MinValue;
            UINT64    MaxValue;
            UINT64    Step;
        } u64;
    } data;
} EFI_IFR_ONE_OF;
```

Members

- **Header**: The sequence that defines the type of opcode as well as the length of the opcode being defined. `Header.OpCode = EFI_IFR_ONE_OF_OP`.
- **Question**: The standard question header. See [Section 33.3.8.2.2](#) for more information.
- **Flags**: Specifies flags related to the numeric question. See “Related Definitions” in **EFI_IFR_NUMERIC**.
- **MinValue**: The minimum value to be accepted by the browser for this opcode. The size of the data field may vary from 8 to 64 bits, depending on the size specified in **Flags**.
MaxValue

The maximum value to be accepted by the browser for this opcode. The size of the data field may vary from 8 to 64 bits, depending on the size specified in Flags.

Step

Defines the amount to increment or decrement the value each time a user requests a value change. If the step value is 0, then the input mechanism for the numeric value is to be free-form and require the user to type in the actual value. The size of the data field may vary from 8 to 64 bits, depending on the size specified in Flags.

Description

This opcode creates a select-on-of object, where the user must select from one of the nested options. This is identical to EFI_IFR_NUMERIC.

There are two ways to specify defaults for this question: one or more nested EFI_IFR_ONE_OF_OPTION (lowest priority) or nested EFI_IFR_DEFAULT (highest priority). An image may be associated with the option using a nested EFI_IFR_IMAGE. An animation may be associated with the question using a nested EFI_IFR_ANIMATION.

33.3.8.3.51 EFI_IFR_ONE_OF_OPTION

Summary

Creates a pre-defined option for a question.

Prototype

```c
#define EFI_IFR_ONE_OF_OPTION_OP 0x09
typedef struct _EFI_IFR_ONE_OF_OPTION {
  EFI_IFR_OP_HEADER     Header;
  EFI_STRING_ID         Option;
  UINT8                 Flags;
  UINT8                 Type;
  EFI_IFR_TYPE_VALUE    Value;
} EFI_IFR_ONE_OF_OPTION;
```

Members

**Header**

The sequence that defines the type of opcode as well as the length of the opcode being defined. `Header.OpCode = EFI_IFR_ONE_OF_OPTION_OP`.

**Option**

The string token reference to the option description string for this particular opcode.

**Flags**

Specifies the flags associated with the current option. See EFI_IFR_OPTION_x.

**Type**

Specifies the type of the option’s value. See EFI_IFR_TYPE.

**Value**

The union of all of the different possible values. The actual contents (and size) of the field depends on Type.
Related Definitions

typedef union {
    UINT8 u8; // EFI_IFR_TYPE_NUM_SIZE_8
    UINT16 u16; // EFI_IFR_TYPE_NUM_SIZE_16
    UINT32 u32; // EFI_IFR_TYPE_NUM_SIZE_32
    UINT64 u64; // EFI_IFR_TYPE_NUM_SIZE_64
    BOOLEAN b; // EFI_IFR_TYPE_BOOLEAN
    EFI_HII_TIME time; // EFI_IFR_TYPE_TIME
    EFI_HII_DATE date; // EFI_IFR_TYPE_DATE
    EFI_STRING_ID string; // EFI_IFR_TYPE_STRING, EFI_IFR_TYPE_ACTION
    EFI_HII_REF ref; // EFI_IFR_TYPE_REF
    // UINT8 buffer[]; // EFI_IFR_TYPE_BUFFER
} EFI_IFR_TYPE_VALUE;

typedef struct {
    UINT8 Hour;
    UINT8 Minute;
    UINT8 Second;
} EFI_HII_TIME;

typedef struct {
    UINT16 Year;
    UINT8 Month;
    UINT8 Day; //
} EFI_HII_DATE;

typedef struct {
    EFI_QUESTION_ID QuestionId;
    EFI_FORM_ID FormId;
    EFI_GUID FormSetGuid;
    EFI_STRING_ID DevicePath;
} EFI_HII_REF;

#define EFI_IFR_TYPE_NUM_SIZE_8  0x00
#define EFI_IFR_TYPE_NUM_SIZE_16 0x01
#define EFI_IFR_TYPE_NUM_SIZE_32 0x02
#define EFI_IFR_TYPE_NUM_SIZE_64 0x03
#define EFI_IFR_TYPE_BOOLEAN    0x04
#define EFI_IFR_TYPE_TIME     0x05
#define EFI_IFR_TYPE_DATE     0x06
#define EFI_IFR_TYPE_STRING    0x07
#define EFI_IFR_TYPE_OTHER     0x08
#define EFI_IFR_TYPE_UNDEF INDED 0x09
#define EFI_IFR_TYPE_ACTION    0x0A
#define EFI_IFR_TYPE_BUFFER    0x0B
#define EFI_IFR_TYPE_REF 0x0C

#define EFI_IFR_OPTION_DEFAULT 0x10
#define EFI_IFR_OPTION_DEFAULT_MFG 0x20

Description
Create a selection for use in any of the questions.
The value is encoded within the opcode itself, unless EFI_IFR_TYPE_OTHER is specified, in which case the value is determined by a nested EFI_IFR_VALUE.

An image may be associated with the option using a nested EFI_IFR_IMAGE. An animation may be associated with the question using a nested EFI_IFR_ANIMATION.

33.3.8.3.52 EFI_IFR_OR

Summary
Pop two Booleans, push TRUE if either is TRUE. Otherwise push FALSE.

Prototype

#define EFI_IFR_OR_OP 0x16
typedef struct _EFI_IFR_OR {
    EFI_IFR_OP_HEADER Header;
} EFI_IFR_OR;

Members

Header The sequence that defines the type of opcode as well as the length of the opcode being defined. Header.OpCode = EFI_IFR_OR_OP.

Description
This opcode performs the following actions:

1. Pop two values from the expression stack.
2. If either value does not evaluate as a Boolean, then push Undefined.
3. If either value evaluates to TRUE, then push TRUE. Otherwise, push FALSE.

33.3.8.3.53 EFI_IFR_ORDERED_LIST

Summary
Creates a set question using an ordered list.
Prototype

```c
#define EFI_IFR_ORDERED_LIST_OP 0x23

typedef struct _EFI_IFR_ORDERED_LIST {
  EFI_IFR_OP_HEADER   Header;
  EFI_IFR_QUESTION_HEADER Question;
  UINT8       MaxContainers;
  UINT8       Flags;
} EFI_IFR_ORDERED_LIST;
```

Members

- **Header**: The byte sequence that defines the type of opcode as well as the length of the opcode being defined. `Header.OpCode = EFI_IFR_ORDERED_LIST_OP`.
- **Question**: The standard question header. See Section 33.3.8.2.2 for more information.
- **MaxContainers**: The maximum number of entries for which this tag will maintain an order. This value also identifies the size of the storage associated with this tag’s ordering array.
- **Flags**: A bit-mask that determines which unique settings are active for this opcode.

Description

Create an ordered list question in the current form. One thing to note is that valid values for the options in ordered lists should never be a 0. The value of 0 is used to determine if a particular "slot" in the array is empty. Therefore, if in the previous example 3 was followed by a 4 and then followed by a 0, the valid options to be displayed would be 3 and 4 only.

An image may be associated with the option using a nested `EFI_IFR_IMAGE`. An animation may be associated with the question using a nested `EFI_IFR_ANIMATION`.

Related Definitions

```c
#define EFI_IFR_UNIQUE_SET   0x01
#define EFI_IFR_NO_EMPTY_SET 0x02
```

These flags determine whether all entries in the list must be unique (`EFI_IFR_UNIQUE_SET`) and whether there can be any empty items in the ordered list (`EFI_IFR_NO_EMPTY_SET`).

33.3.8.3.54 EFI_IFR_PASSWORD

Summary

Creates a password question
Prototype

```c
#define EFI_IFR_PASSWORD_OP 0x08
typedef struct _EFI_IFR_PASSWORD {
    EFI_IFR_OP_HEADER header;
    EFI_IFR_QUESTION_HEADER question;
    UINT16 minsize;
    UINT16 maxsize;
} EFI_IFR_PASSWORD;
```

Members

- **Header**: The sequence that defines the type of opcode as well as the length of the opcode being defined. `header.opcode = EFI_IFR_PASSWORD_OP`.
- **Question**: The standard question header. See [Section 33.3.8.2.2](#) for more information.
- **MinSize**: The minimum number of characters that can be accepted for this opcode.
- **MaxSize**: The maximum number of characters that can be accepted for this opcode.

Description

Creates a password question in the current form.

An image may be associated with the option using a nested `EFI_IFR_IMAGE`. An animation may be associated with the question using a nested `EFI_IFR_ANIMATION`. The password question has two modes of operation. The first is when the `header.flags` has the `EFI_IFR_FLAG_CALLBACK` bit not set. If the bit isn't set, the browser will handle all password operations itself, including string comparisons as needed. If the password question has the `EFI_IFR_FLAG_CALLBACK` bit set, then there will be a formal handshake initiated between the browser and the registered driver that would accept the callback. See the flowchart represented in Figure 33-48 and Figure 33-49 for details.

(This flowchart is provided in two parts because of page formatting but should be viewed as a single continuous chart.)
User selects a password operation in the browser

Browser

Does Password Opcode have the INTERACTIVE bit set?

Yes

Browser calls the ConfigAccess.Callback interface with the Password opcode’s QuestionId. Check for an existing password by sending a NULL string value.

Driver

Is there a preexisting password?

Yes

Driver returns EFI_SUCCESS to indicate there is no preexisting password.

No

Driver will return EFI_SUCCESS if password was accepted and saved.

Browser prompts for the new password and calls ConfigAccess.Callback with the new password string value.

Driver returns EFI_NOT_SUCCESS if password was accepted and saved.

Browser prompts for the existing password and calls ConfigAccess.Callback with the old password string value.

Driver

Did the user type the correct preexisting password?

No

Driver returns an EFI_NOT_READY, indicating the user mistyped the previous password. The browser can optionally alert the user of the error.

Yes

Driver returns EFI_SUCCESS to indicate the user typed the correct preexisting password and wants the user to type a new password.

Driver returns an error other than EFI_NOT_READY, indicating that the browser must exit the password handshake and refresh the current displayed form.

No

Driver returns EFI_SUCCESS if password was accepted and saved.

Yes

Driver returns EFI_SUCCESS to indicate the user typed the correct preexisting password and wants the user to type a new password.

No

Driver returns EFI_SUCCESS to indicate the user typed the correct preexisting password and wants the user to type a new password.

CONTINUED in part two

Figure 33-48 Password Flowchart (part one)
Figure 33-49 Password Flowchart (part two)

**33.3.8.3.55 EFI_IFR_QUESTION_REF1**

**Summary**

Push a question’s value on the expression stack.
Prototype

```c
#define EFI_IFR_QUESTION_REF1_OP 0x40
typedef struct _EFI_IFR_QUESTION_REF1 {
    EFI_IFR_OP_HEADER Header;
    EFI_QUESTION_ID QuestionId;
} EFI_IFR_QUESTION_REF1;
```

Members

- **Header**
  The byte sequence that defines the type of opcode as well as the length of the opcode being defined. `Header.OpCode = EFI_IFR_QUESTION_REF1_OP`.

- **QuestionId**
  The question’s identifier, which must be unique within the form set.

Description

Push the value of the question specified by `QuestionId` on to the expression stack. If the question’s value cannot be determined or the question does not exist, then push Undefined.

### 33.3.8.3.56 EFI_IFR_QUESTION_REF2

Summary

Pop an integer from the expression stack, convert it to a question id, and push the question value associated with that question id onto the expression stack.

Prototype

```c
#define EFI_IFR_QUESTION_REF2_OP 0x41
typedef struct _EFI_IFR_QUESTION_REF2 {
    EFI_IFR_OP_HEADER Header;
} EFI_IFR_QUESTION_REF2;
```

Members

- **Header**
  The byte sequence that defines the type of opcode as well as the length of the opcode being defined. `Header.OpCode = EFI_IFR_QUESTION_REF2_OP`.

Description

This opcode performs the following actions:

1. Pop an integer from the expression stack
2. Convert it to a question id
3. Push the question value associated with that question id onto the expression stack.

If the popped expression cannot be evaluated as an unsigned integer or the value of the unsigned integer is greater than 0xFFFF, then push Undefined onto the expression stack in step 3. If the value of the question specified by the unsigned integer, after converted to a question id, cannot be determined or the question does not exist, also push Undefined onto the expression stack in step 3.
33.3.8.3.57 EFI_IFR_QUESTION_REF3

Summary
Pop an integer from the expression stack, convert it to a question id, and push the question value associated with that question id onto the expression stack.

Prototype
```
#define EFI_IFR_QUESTION_REF3_OP 0x51
typedef struct _ EFI_IFR_QUESTION_REF3 {
    EFI_IFR_OP_HEADER Header;
} EFI_IFR_QUESTION_REF3;

typedef struct _ EFI_IFR_QUESTION_REF3_2 {
    EFI_IFR_OP_HEADER Header;
    EFI_STRING_ID DevicePath;
} EFI_IFR_QUESTION_REF3_2;

typedef struct _ EFI_IFR_QUESTION_REF3_3 {
    EFI_IFR_OP_HEADER Header;
    EFI_STRING_ID DevicePath;
    EFI_GUID Guid;
} EFI_IFR_QUESTION_REF3_3;
```

Members
- **Header**: The byte sequence that defines the type of opcode as well as the length of the opcode being defined. `Header.OpCode = EFI_IFR_QUESTION_REF3_OP`.
- **DevicePath**: Specifies the text representation of the device path containing the form set where the question is defined. If this is not present or the value is 0 then the device path installed on the `EFI_HANDLE` which was registered with the form set containing the current question is used.
- **Guid**: Specifies the GUID of the form set where the question is defined. If the value is Nil or this field is not present, then the current form set is used (if `DevicePath` is 0) or the only form set attached to the device path specified by `DevicePath` is used. If the value is Nil and there is more than one form set on the specified device path, then the value Undefined will be pushed.

Description
This opcode performs the following actions:
1. Pop an integer from the expression stack
2. Convert it to a question id
3. Push the question value associated with that question id onto the expression stack.
If the popped expression cannot be evaluated as an unsigned integer or the value of the unsigned integer is greater than 0xFFFF, then push Undefined onto the expression stack in step 3. If the value of the question specified by the unsigned integer, after converted to a question id, cannot be determined or the question does not exist, also push Undefined onto the expression stack in step 3.

This version allows question values from other forms to be referenced in expressions.

### 33.3.8.3.58 EFI_IFR_READ

**Summary**

Provides a value for the current question or default.

**Prototype**

```c
#define EFI_IFR_READ_OP 0x2D
typedef struct _EFI_IFR_READ {
    EFI_IFR_OP_HEADER    Header;
} EFI_IFR_READ;
```

**Parameters**

- **Header**
  
  The sequence that defines the type of opcode as well as the length of the opcode being defined. For this tag, `Header.OpCode = EFI_IFR_READ_OP`.

**Description**

After reading the value for the current question (if any storage was specified) and setting the `this` constant (see `EFI_IFR_THIS`), this expression will be evaluated (if present) to return the value. If the `FormId` and `QuestionId` are either both not present, or are both set to zero, then the link does nothing.

### 33.3.8.3.59 EFI_IFR_REF

**Summary**

Creates a cross-reference statement.
Prototype

```
#define EFI_IFR_REF_OP 0x0F
typedef struct _EFI_IFR_REF {
  EFI_IFR_OP_HEADER   Header;
  EFI_IFR_QUESTION_HEADER  Question;
  EFI_FORM_ID          FormId;
} EFI_IFR_REF;

typedef struct _EFI_IFR_REF2 {
  EFI_IFR_OP_HEADER   Header;
  EFI_IFR_QUESTION_HEADER  Question;
  EFI_FORM_ID          FormId;
  EFI_QUESTION_ID      QuestionId;
} EFI_IFR_REF2;

typedef struct _EFI_IFR_REF3 {
  EFI_IFR_OP_HEADER   Header;
  EFI_IFR_QUESTION_HEADER  Question;
  EFI_FORM_ID          FormId;
  EFI_QUESTION_ID      QuestionId;
  EFI_GUID             FormSetId;
} EFI_IFR_REF3;

typedef struct _EFI_IFR_REF4 {
  EFI_IFR_OP_HEADER   Header;
  EFI_IFR_QUESTION_HEADER  Question;
  EFI_FORM_ID          FormId;
  EFI_QUESTION_ID      QuestionId;
  EFI_GUID             FormSetId;
  EFI_STRING_ID        DevicePath;
} EFI_IFR_REF4;

typedef struct _EFI_IFR_REF5 {
  EFI_IFR_OP_HEADER   Header;
  EFI_IFR_QUESTION_HEADER  Question;
} EFI_IFR_REF5;
```

Members

- **Header**
  - The byte sequence that defines the type of opcode as well as the length of the opcode being defined. `Header.OpCode = EFI_IFR_REF_OP`.

- **Question**
  - Standard question header. See Section 33.3.8.2.2

- **FormId**
  - The form to which this link is referring. If this is zero, then the link is on the current form. If this is missing, then the link is determined by the nested `EFI_IFR_VALUE`. 
**QuestionId**
The question on the form to which this link is referring. If this field is not present (determined by the length of the opcode) or the value is zero, then the link refers to the top of the form.

**FormSetId**
The form set to which this link is referring. If it is all zeroes or not present, and `DevicePath` is not present, then the link is to the current form set. If it is all zeroes (or not present) and the `DevicePath` is present, then the link is to the first form set associated with the `DevicePath`.

**DevicePath**
The string identifier that specifies the string containing the text representation of the device path to which the form set containing the form specified by `FormId`.

If this field is not present (determined by the opcode's length) or the value is zero, then the link refers to the current page. The format of the device path string that this field references is compatible with the Text format that is specified in the Text Device Node Reference (Section 10.6.1.6)

**Description**
Creates a user-selectable link to a form or a question on a form. There are several forms of this opcode which are distinguished by the length of the opcode.

### 33.3.8.3.60 EFI_IFR_REFRESH

**Summary**
Mark a question for periodic refresh.

**Prototype**

```c
#define EFI_IFR_REFRESH_OP 0x1d
typedef struct _EFI_IFR_REFRESH {
    EFI_IFR_OP_HEADER Header;
    UINT8 RefreshInterval;
} EFI_IFR_REFRESH;
```

**Members**

- **Header**
The byte sequence that defines the type of opcode as well as the length of the opcode being defined. `Header.OpCode = EFI_IFR_REFRESH_OP`.

- **RefreshInterval**
Minimum number of seconds before the question value should be refreshed. A value of zero indicates the question should not be refreshed automatically.

**Description**
When placed within the scope of a question, it will force the question’s value to be refreshed at least every `RefreshInterval` seconds. The value may be refreshed less often, depending on browser policy or capabilities.
33.3.8.3.61 EFI_IFR_REFRESH_ID

Summary
Mark an Question for an asynchronous refresh.

Prototype
```
#define EFI_IFR_REFRESH_ID_OP 0x62
typedef struct _EFI_IFR_REFRESH_ID {
  EFI_IFR_OP_HEADER Header;
  EFI_GUID RefreshEventGroupId;
} EFI_IFR_REFRESH_ID;
```

Members
- **Header**: The byte sequence that defines the type of opcode as well as the length of the opcode being defined. `Header.OpCode = EFI_IFR_REFRESH_ID_OP`.
- **RefreshEventGroupId**: The GUID associated with the event group which will be used to initiate a re-evaluation of an element in a set of forms.

Description
This tag op-code must be placed within the scope of a question or a form. If within the scope of a question and the event is signaled which belongs to the RefreshEventGroupId, the question will be refreshed. More than one question may share the same Event Group.

If the tag op-code is placed within the scope of an EFI_IFR_FORM op-code and the event is signaled which belongs to the RefreshEventGroupId, the entire form’s contents will be refreshed.

- If the contents within a form had an EFI_IFR_REFRESH_ID tag op-code placed within the scope of the form, and an event is signaled, all questions associated with the RefreshEventGroupId are marked for refresh. The Forms Browser will update the question value from storage, reparse the forms from the HII database and, at some time later, reflect that change if the question is displayed.

When interpreting this op-code, a browser must do the following actions:

- The browser will create an event group via CreateEventEx() based on the specified RefreshEventGroupId when the form set which contains the op-code is opened by the browser.
- When an event is signaled, all questions associated with the RefreshEventGroupId are marked for refresh. The Forms Browser will update the question value from storage and, at some time later, update the question’s display.
- The browser will close the event group which was previously created when the form set which contains the op-code is closed by the browser.
33.3.8.3.62 EFI_IFR_RESET_BUTTON

Summary
Create a reset or submit button on the current form.

Prototype

```
#define EFI_IFR_RESET_BUTTON_OP 0x0d
typedef struct _EFI_IFR_RESET_BUTTON {
    EFI_IFR_OP_HEADER    Header;
    EFI_IFR_STATEMENT_HEADER Statement;
    EFI_DEFAULT_ID    DefaultId;
} EFI_IFR_RESET_BUTTON;

typedef UINT16 EFI_DEFAULT_ID;
```

Members

- **Header**: The standard header, where `Header.OpCode = EFI_IFR_RESET_BUTTON_OP`.
- **Statement**: Standard statement header, including the prompt and help text.
- **DefaultId**: Specifies the set of default store to use when restoring the defaults to the questions on this form. See `EFI_IFR_DEFAULTSTORE` (Section 33.3.8.3.13) for more information.

Description
This opcode creates a user-selectable button that resets the question values for all questions on the current form to the default values specified by `DefaultId`. If `EFI_IFR_FLAGS_CALLBACK` is set in the question header, then the callback associated with the form set will be called. An image may be associated with the statement using a nested `EFI_IFR_IMAGE`. An animation may be associated with the statement using a nested `EFI_IFR_ANIMATION`.

33.3.8.3.63 EFI_IFR_RULE

Summary
Create a rule for use in a form and associate it with an identifier.
Prototype

```c
#define EFI_IFR_RULE_OP 0x18
typedef struct _EFI_IFR_RULE {
    EFI_IFR_OP_HEADER Header;
    UINT8 RuleId;
} EFI_IFR_RULE;
```

Members

- **Header**: The byte sequence that defines the type of opcode as well as the length of the opcode being defined. `Header OpCode = EFI_IFR_RULE_OP`.
- **RuleId**: Unique identifier for the rule. There can only be one rule within a form with the specified `RuleId`. If another already exists, then the form is marked as invalid.

Description

Create a rule, which associates an expression with an identifier and attaches it to the currently opened form. These rules allow common sub-expressions to be re-used within a form.

33.3.8.3.64 EFI_IFR_RULE_REF

Summary

Evaluate a form rule and push its result on the expression stack.

Prototype

```c
#define EFI_IFR_RULE_REF_OP 0x3f
typedef struct _EFI_IFR_RULE_REF {
    EFI_IFR_OP_HEADER Header;
    UINT8 RuleId;
} EFI_IFR_RULE_REF;
```

Members

- **Header**: The byte sequence that defines the type of opcode as well as the length of the opcode being defined. `Header OpCode = EFI_IFR_RULE_REF_OP`.
- **RuleId**: The rule’s identifier, which must be unique within the form.

Description

Look up the rule specified by `RuleId` and push the evaluated result on the expression stack. If the specified rule does not exist, then push `Undefined`.

33.3.8.3.65 EFI_IFR_SECURITY

Summary

Push `TRUE` if the current user profile contains the specified setup access permissions.
### EFI_IFR_SECURITY

#### Prototype

```c
#define EFI_IFR_SECURITY_OP     0x60
typedef struct _EFI_IFR_SECURITY {
  EFI_IFR_OP_HEADER         Header;
  EFI_GUID                  Permissions;
} EFI_IFR_SECURITY;
```

#### Members

- **Header**: Standard opcode header, where `Header.Op = EFI_IFR_SECURITY_OP`.
- **Permissions**: Security permission level.

#### Description

This opcode pushes whether or not the current user profile contains the specified setup access permissions. This opcode can be used in expressions to disable, suppress or gray-out forms, statements and questions. It can also be used in checking question values to disallow or allow certain values.

This opcode performs the following actions:

1. If the current user profile contains the specified setup access permissions, then push `TRUE`. Otherwise, push `FALSE`.

### 33.3.8.3.66 EFI_IFR_SET

#### Summary

Change a stored value.

#### Prototype

```c
#define EFI_IFR_SET_OP 0x2C
typedef struct _EFI_IFR_SET {
  EFI_IFR_OP_HEADER         Header;
  EFI_VARSTORE_ID           VarStoreId;
  union {
    EFI_STRING_ID            VarName;
    UINT16                   VarOffset;
  } VarStoreInfo;
  UINT8                     VarStoreType;
} EFI_IFR_SET;
```

#### Parameters

- **Header**: The sequence that defines the type of opcode as well as the length of the opcode being defined. `Header.OpCode = EFI_IFR_SET_OP`.
- **VarStoreId**: Specifies the identifier of a previously declared variable store to use when storing the question’s value.
VarStoreInfo  Depending on the type of variable store selected, this contains either a 16-bit Buffer Storage offset (VarOffset) or a Name/Value or EFI Variable name (VarName).

VarStoreType  Specifies the type used for storage. The storage types EFI_IFR_TYPE_x are defined in EFI_IFR_ONE_OF_OPTION.

Description
This operator pops an expression from the expression stack. The expression popped is the value. The value is stored into the variable store identified by VarStoreId and VarStoreInfo. If the value could be stored successfully, then TRUE is pushed on to the expression stack. Otherwise, FALSE is pushed on the expression stack.

33.383.67 EFI_IFR_SHIFT_LEFT

Summary
Pop two unsigned integers, shift one left by the number of bits specified by the other and push the result.

Prototype
#define EFI_IFR_SHIFT_LEFT_OP 0x38
typedef struct _EFI_IFR_SHIFT_LEFT {
  EFI_IFR_OP_HEADER Header;
} EFI_IFR_SHIFT_LEFT;

Members
Header Standard opcode header, where OpCode is EFI_IFR_SHIFT_LEFT_OP.

Description
This opcode performs the following actions:
1. Pop two values from the expression stack. The first value popped is the right-hand value and the second value popped is the left-hand value.
2. If the two values do not evaluate to unsigned integers, push Undefined.
3. Shift the left-hand value left by the number of bits specified by the right-hand value and push the result.

33.383.68 EFI_IFR_SHIFT_RIGHT

Summary
Pop two unsigned integers, shift one right by the number of bits specified by the other and push the result.
Prototype

```c
#define EFI_IFR_SHIFT_RIGHT_OP 0x39
typedef struct _EFI_IFR_SHIFT_RIGHT {
    EFI_IFR_OP_HEADER Header;
} EFI_IFR_SHIFT_RIGHT;
```

Members

- **Header**: Standard opcode header, where OpCode is EFI_IFR_SHIFT_RIGHT_OP.

Description

This opcode performs the following actions:

1. Pop two values from the expression stack. The first value popped is the right-hand value and the second value popped is the left-hand value.
2. If the two values do not evaluate to unsigned integers, push Undefined.
3. Shift the left-hand value right by the number of bits specified by the right-hand value and push the result.

### 33.3.8.3.69 EFI_IFR_SPAN

Summary

Pop two strings and an unsigned integer, find the first character from one string that contains characters found in another and push its index.

Prototype

```c
#define EFI_IFR_SPAN_OP 0x59
typedef struct _EFI_IFR_SPAN {
    EFI_IFR_OP_HEADER Header;
    UINT8 Flags;
} EFI_IFR_SPAN;
```

Members

- **Header**: The sequence that defines the type of opcode as well as the length of the opcode being defined. For this tag, Header.OpCode = EFI_IFR_SPAN_OP.
- **Flags**: Specifies whether to find the first matching string (EFI_IFR_FLAGS_FIRST_MATCHING) or the first non-matching string (EFI_IFR_FLAGS_FIRST_NON_MATCHING).

Description

This opcode performs the following actions:

1. Pop three values from the expression stack. The first value popped is the right value and the second value popped is the middle value and the last value popped is the left expression.
2. If the left or middle values cannot be evaluated as a string, push Undefined. If the right value cannot be evaluated as an unsigned integer, push Undefined.
3. The left string is the string to scan. The middle string consists of character pairs representing the low-end of a range and the high-end of a range of characters. The right unsigned integer represents the starting location for the scan.

4. The operation will push the zero-based index of the first character after the right value which falls within any one of the ranges (EFI_IFR_FLAGS_FIRST_MATCHING) or falls within none of the ranges (EFI_IFR_FLAGS_FIRST_NON_MATCHING).

Related Definitions

#define EFI_IFR_FLAGS_FIRST_MATCHING    0x00
#define EFI_IFR_FLAGS_FIRST_NON_MATCHING  0x01

33.3.8.3.70 EFI_IFR_STRING

Summary

Defines the string question.

Prototype

#define EFI_IFR_STRING_OP 0x1C
typedef struct _EFI_IFR_STRING {
    EFI_IFR_OP_HEADER       Header;
    EFI_IFR_QUESTION_HEADER Question;
    UINT8                   MinSize;
    UINT8                   MaxSize;
    UINT8                   Flags;
} EFI_IFR_STRING;

Members

Header                The sequence that defines the type of opcode as well as the length of the opcode being defined. Header.OpCode = EFI_IFR_STRING_OP.
Question              The standard question header. See Section 33.3.8.2.2 for more information.
MinSize               The minimum number of characters that can be accepted for this opcode.
MaxSize               The maximum number of characters that can be accepted for this opcode.
Flags                 Flags which control the string editing behavior. See “Related Definitions” below.

Description

This creates a string question. The minimum length is MinSize and the maximum length is MaxSize characters.

An image may be associated with the question using a nested EFI_IFR_IMAGE. An animation may be associated with the question using a nested EFI_IFR_ANIMATION.

There are two ways to specify defaults for this question: one or more nested EFI_IFR_ONE_OF_OPTION (lowest priority) or nested EFI_IFR_DEFAULT (highest priority).
If `EFI_IFR_STRING_MULTI_LINE` is set, it is a hint to the Forms Browser that multi-line text can be allowed. If it is clear, then multi-line editing should not be allowed.

Related Definitions

```c
#define EFI_IFR_STRING_MULTI_LINE 0x01
```

33.3.8.3.71 EFI_IFR_STRING_REF1

Summary

Push a string on the expression stack.

Prototype

```c
#define EFI_IFR_STRING_REF1_OP 0x4e
typedef struct _EFI_IFR_STRING_REF1 {
    EFI_IFR_OP_HEADER Header;
    EFI_STRING_ID StringId;
} EFI_IFR_STRING_REF1;
```

Members

- **Header**
  The byte sequence that defines the type of opcode as well as the length of the opcode being defined. `Header.OpCode = EFI_IFR_STRING_REF1_OP`.

- **StringId**
  The string’s identifier, which must be unique within the package list.

Description

Push the string specified by `StringId` on to the expression stack. If the string does not exist, then push an empty string.

33.3.8.3.72 EFI_IFR_STRING_REF2

Summary

Pop a string identifier, push the associated string.

Prototype

```c
#define EFI_IFR_STRING_REF2_OP 0x4f
typedef struct _EFI_IFR_STRING_REF2 {
    EFI_IFR_OP_HEADER Header;
} EFI_IFR_STRING_REF2;
```

Members

- **Header**
  The byte sequence that defines the type of opcode as well as the length of the opcode being defined. `Header.OpCode = EFI_IFR_STRING_REF2_OP`.

Description

This opcode performs the following actions:
1. Pop a value from the expression stack.
2. If the value cannot be evaluated as an unsigned integer or the value of the unsigned integer is greater than 0xFFFF, push Undefined.
3. If the string specified by the value (converted to a string identifier) cannot be determined or the string does not exist, push an empty string.
4. Otherwise, push the string on to the expression stack.

33.3.8.3.73 EFI_IFR_SUBTITLE

Summary
Creates a sub-title in the current form.

Prototype
#define EFI_IFR_SUBTITLE_OP 0x02
typedef struct _EFI_IFR_SUBTITLE {
  EFI_IFR_OP_HEADER     Header;
  EFI_IFR_STATEMENT_HEADER Statement;
  UINT8                Flags;
} EFI_IFR_SUBTITLE;

Members

  Header       The sequence that defines the type of opcode as well as the length of the opcode being defined. For this tag, Header.OpCode = EFI_IFR_SUBTITLE_OP.
  Flags        Identifies specific behavior for the sub-title.

Description
Subtitle strings are intended to be used by authors to separate sections of questions into semantic groups. If Header.Scope is set, then the Forms Browser may further distinguish the end of the semantic group as including only those statements and questions which are nested.

If EFI_IFR_FLAGS_HORIZONTAL is set, then this provides a hint that the nested statements or questions should be horizontally arranged. Otherwise, they are assumed to be vertically arranged.

An image may be associated with the statement using a nested EFI_IFR_IMAGE. An animation may be associated with the statement using a nested EFI_IFR_ANIMATION.

Related Definitions
#define EFI_IFR_FLAGS_HORIZONTAL 0x01

33.3.8.3.74 EFI_IFR_SUBTRACT

Summary
Pop two unsigned integers, subtract one from the other, push the result.
Prototype

```c
#define EFI_IFR_SUBTRACT_OP 0x3b
typedef struct _EFI_IFR_SUBTRACT {
    EFI_IFR_OP_HEADER Header;
} EFI_IFR_SUBTRACT;
```

Members

- **Header**

  Standard opcode header, where `Header.OpCode` is `EFI_IFR_SUBTRACT_OP`.

Description

This opcode performs the following operations:

1. Pop two values from the expression stack. The first value popped is the *right-hand* value and the second value popped is the *left-hand* value.
2. If the two values do not evaluate to unsigned integers, push Undefined.
3. Zero-extend the values to 64-bits.
4. Subtract the *right-hand* value from the *left-hand* value.
5. Push the lower 64-bits of the result.

### 33.3.8.3.75 EFI_IFR_SUPPRESS_IF

Summary

Creates a group of statements or questions which are conditionally invisible.

Prototype

```c
#define EFI_IFR_SUPPRESS_IF_OP 0x0a
typedef struct _EFI_IFR_SUPPRESS_IF {
    EFI_IFR_OP_HEADER Header;
} EFI_IFR_SUPPRESS_IF;
```

Members

- **Header**

  The byte sequence that defines the type of opcode as well as the length of the opcode being defined. `Header.OpCode` is `EFI_IFR_SUPPRESS_IF_OP`.

Description

The suppress tag causes the nested objects to be hidden from the user if the expression appearing as the first nested object evaluates to TRUE. If the expression consists of more than a single opcode, then the first opcode in the expression must have the Scope bit set and the expression must end with `EFI_IFR_END`.

This display form is maintained until the scope for this opcode is closed.
### 33.3.8.3.76 EFI_IFR_TEXT

**Summary**
Creates a static text and image.

**Prototype**
```
#define EFI_IFR_TEXT_OP 0x03
typedef struct _EFI_IFR_TEXT {
    EFI_IFR_OP_HEADER Header;
    EFI_IFR_STATEMENT_HEADER Statement;
    EFI_STRING_ID TextTwo;
} EFI_IFR_TEXT;
```

**Members**
- **Header**
  The sequence that defines the type of opcode as well as the length of the opcode being defined. For this tag, `Header.OpCode = EFI_IFR_TEXT_OP`.
- **Statement**
  Standard statement header.
- **TextTwo**
  The string token reference to the secondary string for this opcode.

**Description**
This is a static text/image statement.

An image may be associated with the statement using a nested `EFI_IFR_IMAGE`. An animation may be associated with the question using a nested `EFI_IFR_ANIMATION`.

### 33.3.8.3.77 EFI_IFR_THIS

**Summary**
Push current question’s value.

**Prototype**
```
#define EFI_IFR_THIS_OP 0x58
typedef struct _EFI_IFR_THIS {
    EFI_IFR_OP_HEADER Header;
} EFI_IFR_THIS;
```

**Members**
- **Header**
  The sequence that defines the type of opcode as well as the length of the opcode being defined. For this tag, `Header.OpCode = EFI_IFR_THIS_OP`.

**Description**
Push the current question’s value.
33.3.8.3.78 EFI_IFR_TIME

Summary
Create a Time question.

Prototype
```
#define EFI_IFR_TIME_OP 0x1b
typedef struct _EFI_IFR_TIME {
    EFI_IFR_OP_HEADER Header;
    EFI_IFR_QUESTION_HEADER Question;
    UINT8 Flags;
} EFI_IFR_TIME;
```

Members
- **Header**: Basic question information. `Header.Opcode = EFI_IFR_TIME_OP`.
- **Question**: The standard question header. See Section 33.3.8.2.2 for more information.
- **Flags**: A bit-mask that determines which unique settings are active for this opcode.

  - `QF_TIME_HOUR_SUPPRESS` 0x01
  - `QF_TIME_MINUTE_SUPPRESS` 0x02
  - `QF_TIME_SECOND_SUPPRESS` 0x04
  - `QF_TIME_STORAGE` 0x30

  For `QF_TIME_STORAGE`, there are currently three valid values:

  - `QF_TIME_STORAGE_NORMAL` 0x00
  - `QF_TIME_STORAGE_TIME` 0x10
  - `QF_TIME_STORAGE_WAKEUP` 0x20

Description
Create a Time question (see Section 33.2.5.4.11) and add it to the current form.

An image may be associated with the question using a nested `EFI_IFR_IMAGE`. An animation may be associated with the question using a nested `EFI_IFR_ANIMATION`.

33.3.8.3.79 EFI_IFR_TOKEN

Summary
Pop two strings and an unsigned integer, then push the nth section of the first string using delimiters from the second string.
Prototype

```c
#define EFI_IFR_TOKEN_OP 0x4d
typedef struct _EFI_IFR_TOKEN {
  EFI_IFR_OP_HEADER    Header;
} EFI_IFR_TOKEN;
```

Members

- **Header**: Standard opcode header, where **OpCode** is **EFI_IFR_TOKEN_OP**.

Description

This opcode performs the following actions:

1. Pop three values from the expression stack. The first value popped is the **right** value and the second value popped is the **middle** value and the last value popped is the **left** value.
2. If the **left** or **middle** values cannot be evaluated as a string, push Undefined. If the **right** value cannot be evaluated as an unsigned integer, push Undefined.
3. The first value is the string. The second value is a string, where each character is a valid delimiter. The third value is the zero-based index.
4. Push the nth delimited sub-string on to the expression stack (0 = left of the first delimiter). The end of the string always acts a the final delimiter.
5. The no such string exists, an empty string is pushed.

### 33.3.8.3.80 EFI_IFR_TO_BOOLEAN

Summary

Pop a value, convert to Boolean and push the result.

Prototype

```c
#define EFI_IFR_TO_BOOLEAN_OP 0x4A
typedef struct _EFI_IFR_TO_BOOLEAN {
  EFI_IFR_OP_HEADER    Header;
} EFI_IFR_TO_BOOLEAN;
```

Members

- **Header**: The sequence that defines the type of opcode as well as the length of the opcode being defined. For this tag, **Header.OpCode** = **EFI_IFR_TO_BOOLEAN_OP**

Description

This opcode performs the following actions:

1. Pop a value from the expression stack. If the value is Undefined or cannot be evaluated as a Boolean, push Undefined. Otherwise push the Boolean on the expression stack.
2. When converting from an unsigned integer, zero will be converted to \texttt{FALSE} and any other value will be converted to \texttt{TRUE}.

3. When converting from a string, if case-insensitive compare with “true” is \texttt{TRUE}, then push \texttt{TRUE}. If a case-insensitive compare with “false” is \texttt{TRUE}, then push \texttt{False}. Otherwise, push Undefined.

4. When converting from a buffer, if the buffer is all zeroes, then push \texttt{False}. Otherwise push \texttt{True}.

\subsection*{33.3.8.3.81 EFI_IFR_TO_LOWER}

**Summary**

Convert a string on the expression stack to lower case.

**Prototype**

```c
#define EFI_IFR_TO_LOWER_OP 0x20
typedef struct _EFI_IFR_TO_LOWER {
  EFI_IFR_OP_HEADER Header;
} EFI_IFR_TO_LOWER;
```

**Members**

- \textit{Header}
  
The sequence that defines the type of opcode as well as the length of the opcode being defined. For this tag, \texttt{Header.OpCode = EFI_IFR_TO_LOWER_OP}

**Description**

Pop an expression from the expression stack. If the expression is Undefined or cannot be evaluated as a string, push Undefined. Otherwise, convert the string to all lower case using the \texttt{StrLwr} function of the \texttt{EFI_UNICODE_COLLATION2_PROTOCOL} and push the string on the expression stack.

\subsection*{33.3.8.3.82 EFI_IFR_TO_STRING}

**Summary**

Pop a value, convert to a string, push the result.

**Prototype**

```c
#define EFI_IFR_TO_STRING_OP 0x49
typedef struct _EFI_IFR_TO_STRING{
  EFI_IFR_OP_HEADER Header;
  UINT8 Format;
} EFI_IFR_TO_STRING;
```

**Members**

- \textit{Header}
  
The sequence that defines the type of opcode as well as the length of the opcode being defined. For this tag, \texttt{Header.OpCode = EFI_IFR_TO_STRING_OP}

- \textit{Format}
  
  When converting from unsigned integers, these flags control the format:
0 = unsigned decimal
1 = signed decimal
2 = hexadecimal (lower-case alpha)
3 = hexadecimal (upper-case alpha)
When converting from a buffer, these flags control the format:
0 = ASCII
8 = UCS-2

Description
This opcode performs the following actions:

1. Pop a value from the expression stack.
2. If the value is Undefined or cannot be evaluated as a string, push Undefined.
3. Convert the value to a string. When converting from an unsigned integer, the number will be converted to an unsigned decimal string (Format = 0), signed decimal string (Format = 1) or a hexadecimal string (Format = 2 or 3).
   When converting from a boolean, the boolean will be converted to “True” (True) or “False” (False). When converting from a buffer, each 8-bit (Format =0) or 16-bit (Format =8) value will be converted into a character and appended to the string, up until the end of the buffer or a NULL character.
4. Push the result.

33.3.8.3.83 EFI_IFR_TO_UINT

Summary
Pop a value, convert to an unsigned integer, push the result.

Prototype
#define EFI_IFR_TO_UINT_OP 0x48
typedef struct _EFI_IFR_TO_UINT {
   EFI_IFR_OP_HEADER Header;
} EFI_IFR_TO_UINT;

Members
Header The sequence that defines the type of opcode as well as the length of the opcode being defined. For this tag,
Header .OpCode = EFI_IFR_TO_UINT_OP

Description
1. Pop a value from the expression stack.
2. If the value is Undefined or cannot be evaluated as an unsigned integer, push Undefined.
3. Convert the value to an unsigned integer. When converting from a boolean, if True, push 1 and if False, push 0. When converting from a string, whitespace is skipped. The prefix ‘0x’ or ‘0X’ indicates to convert from a hexadecimal string while the prefix ‘-‘ indicates conversion from a signed integer string. When converting from a buffer, if the buffer is greater than 8 bytes in length, push Undefined. Otherwise, zero-extend the contents of the buffer to 64-bits.
4. Push the result.

### 33.3.8.3.84 EFI_IFR_TO_UPPER

**Summary**
Convert a string on the expression stack to upper case.

**Prototype**
```c
#define EFI_IFR_TO_UPPER_OP 0x21
typedef struct _EFI_IFR_TO_UPPER {
    EFI_IFR_OP_HEADER  Header;
} EFI_IFR_TO_UPPER;
```

**Members**
- **Header**
  The sequence that defines the type of opcode as well as the length of the opcode being defined. For this tag, `Header.OpCode = EFI_IFR_TO_UPPER_OP`

**Description**
Pop an expression from the expression stack. If the expression is Undefined or cannot be evaluated as a string, push Undefined. Otherwise, convert the string to all upper case using the `StrUp` function of the `EFI_UNICODE_COLLATION2_PROTOCOL` and push the string on the expression stack.

### 33.3.8.3.85 EFI_IFR_TRUE

**Summary**
Push a TRUE on to the expression stack.

**Prototype**
```c
#define EFI_IFR_TRUE_OP 0x46
typedef struct _EFI_IFR_TRUE {
    EFI_IFR_OP_HEADER  Header;
} EFI_IFR_TRUE;
```

**Members**
- **Header**
  The sequence that defines the type of opcode as well as the length of the opcode being defined. For this tag, `Header.OpCode = EFI_IFR_TRUE_OP`

**Description**
Push a TRUE on to the expression stack.

### 33.3.8.3.86 EFI_IFR_UINT8, EFI_IFR_UINT16, EFI_IFR_UINT32, EFI_IFR_UINT64

**Summary**
Push an unsigned integer on to the expression stack.
Prototype

```c
#define EFI_IFR_UINT8_OP 0x42
typedef struct _EFI_IFR_UINT8 {
    EFI_IFR_OP_HEADER Header;
    UINT8 Value;
} EFI_IFR_UINT8;

#define EFI_IFR_UINT16_OP 0x43
typedef struct _EFI_IFR_UINT16 {
    EFI_IFR_OP_HEADER Header;
    UINT16 Value;
} EFI_IFR_UINT16;

#define EFI_IFR_UINT32_OP 0x44
typedef struct _EFI_IFR_UINT32 {
    EFI_IFR_OP_HEADER Header;
    UINT32 Value;
} EFI_IFR_UINT32;

#define EFI_IFR_UINT64_OP 0x45
typedef struct _EFI_IFR_UINT64 {
    EFI_IFR_OP_HEADER Header;
    UINT64 Value;
} EFI_IFR_UINT64;
```

Members

- **Header**: The sequence that defines the type of opcode as well as the length of the opcode being defined. For this tag, `Header.OpCode = EFI_IFR_UINT8_OP, EFI_IFR_UINT16_OP, EFI_IFR_UINT32_OP` or `EFI_IFR_UINT64_OP`.
- **Value**: The unsigned integer.

Description

Push the specified unsigned integer, zero-extended to 64-bits, on to the expression stack.

### 33.3.8.3.87 EFI_IFR_UNDEFINED

**Summary**

Push an Undefined to the expression stack.
Prototype

```c
#define EFI_IFR_UNDEFINED_OP 0x55
typedef struct _EFI_IFR_UNDEFINED {
    EFI_IFR_OP_HEADER Header;
} EFI_IFR_UNDEFINED;
```

Members

- **Header**
  
  The sequence that defines the type of opcode as well as the length of the opcode being defined. For this tag, `Header.OpCode = EFI_IFR_UNDEFINED_OP`

Description

Push Undefined on to the expression stack.

### 33.3.8.3.88 EFI_IFR_VALUE

**Summary**

Provides a value for the current question or default.

Prototype

```c
#define EFI_IFR_VALUE_OP 0x5a
typedef struct _EFI_IFR_VALUE {
    EFI_IFR_OP_HEADER Header;
} EFI_IFR_VALUE;
```

Members

- **Header**
  
  The sequence that defines the type of opcode as well as the length of the opcode being defined. For this tag, `Header.OpCode = EFI_IFR_VALUE_OP`

Description

Creates a value for the current question or default with no storage. The value is the result of the expression nested in the scope.

If used for a question, then the question will be read-only.

### 33.3.8.3.89 EFI_IFR_VARSTORE

**Summary**

Creates a variable storage short-cut for linear buffer storage.
Prototype

```c
#define EFI_IFR_VARSTORE_OP 0x24
typedef struct {
    EFI_IFR_OP_HEADER Header;
    EFI_GUID Guid;
    EFI_VARSTORE_ID VarStoreId;
    UINT16 Size;
    //UINT8 Name[];
} EFI_IFR_VARSTORE;
```

Members

- **Header**: The byte sequence that defines the type of opcode as well as the length of the opcode being defined. For this tag, `Header.OpCode = EFI_IFR_VARSTORE_OP`.
- **Guid**: The variable’s GUID definition. This field comprises one half of the variable name, with the other half being the human-readable aspect of the name, which is represented by the string immediately following the Size field. Type `EFI_GUID` is defined in `InstallProtocolInterface()` in this specification.
- **VarStoreId**: The variable store identifier, which is unique within the current form set. This field is the value that uniquely identify this instance from others. Question headers refer to this value to designate which is the active variable that is being used. A value of zero is invalid.
- **Size**: The size of the variable store.
- **Name**: A null-terminated ASCII string that specifies the name associated with the variable store. The field is not actually included in the structure but is included here to help illustrate the encoding of the opcode. The size of the string, including the null termination, is included in the opcode’s header size.

Description

This opcode describes a Buffer Storage Variable Store within a form set. A question can select this variable store by setting the `VarStoreId` field in its opcode header.

An `EFI_IFR_VARSTORE` with a specified `VarStoreId` must appear in the IFR before it can be referenced by a question.

### 33.3.8.3.90 EFI_IFR_VARSTORE_NAME_VALUE

Summary

Creates a variable storage short-cut for name/value storage.
Prototype

```c
#define EFI_IFR_VARSTORE_NAME_VALUE_OP 0x25
typedef struct _EFI_IFR_VARSTORE_NAME_VALUE {
    EFI_IFR_OP_HEADER Header;
    EFI_VARSTORE_ID VarStoreId;
    EFI_GUID Guid;
} EFI_IFR_VARSTORE_NAME_VALUE;
```

Members

- **Header**: The byte sequence that defines the type of opcode as well as the length of the opcode being defined. For this tag, `Header.OpCode = EFI_IFR_VARSTORE_NAME_VALUE_OP`.
- **Guid**: The variable's GUID definition. This field comprises one half of the variable name, with the other half being the human-readable aspect of the name, which is specified in the `VariableName` field in the question's header (see `EFI_IFR_QUESTION_HEADER`). Type `EFI_GUID` is defined in `InstallProtocolInterface()` in the UEFI Specification.
- **VarStoreId**: The variable store identifier, which is unique within the current form set. This field is the value that uniquely identifies this variable store definition instance from others. Question headers refer to this value to designate which is the active variable that is being used. A value of zero is invalid.

Description

This opcode describes a Name/Value Variable Store within a form set. A question can select this variable store by setting the `VarStoreId` field in its question header.

An `EFI_IFR_VARSTORE_NAME_VALUE` with a specified `VarStoreId` must appear in the IFR before it can be referenced by a question.

33.3.8.3.91 EFI_IFR_VARSTORE_EFI

Summary

Creates a variable storage short-cut for EFI variable storage.
Prototype

```c
#define EFI_IFR_VARSTORE_EFI_OP 0x26
typedef struct _EFI_IFR_VARSTORE_EFI {
    EFI_IFR_OP_HEADER Header;
    EFI_VARSTORE_ID VarStoreId;
    EFI_GUID Guid;
    UINT32 Attributes;
    UINT16 Size;
    //UINT8 Name[];
} EFI_IFR_VARSTORE_EFI;
```

Members

- **Header**
  The byte sequence that defines the type of opcode as well as the length of the opcode being defined. For this tag, `Header.OpCode = EFI_IFR_VARSTORE_EFI_OP`.

- **VarStoreId**
  The variable store identifier, which is unique within the current form set. This field is the value that uniquely identifies this variable store definition instance from others. Question headers refer to this value to designate which is the active variable that is being used. A value of zero is invalid.

- **Guid**
  The EFI variable's GUID definition. This field comprises one half of the EFI variable name, with the other half being the human-readable aspect of the name, which is specified in the `Name` field below. Type `EFI_GUID` is defined in `InstallProtocolInterface()` in this specification.

- **Attributes**
  Specifies the flags to use for the variable.

- **Size**
  The size of the variable store.

- **Name**
  A null-terminated ASCII string that specifies one half of the EFI name for this variable store. The other half is specified in the `Guid` field (above). The `Name` field is not actually included in the structure but is included here to help illustrate the encoding of the opcode. The size of the string, including the null termination, is included in the opcode's header size.

Description

This opcode describes an EFI Variable Variable Store within a form set. The `Guid` and `Name` specified here will be used with `GetVariable()` and `SetVariable()`.

- A question can select this variable store by setting the `VarStoreId` field in its question header.
- A question can refer to a specific offset within the EFI Variable using the `VarOffset` field in its question header.
Note: Name must be converted to a CHAR16 string before it is passed to GetVariable() or SetVariable().

An EFI_IFR_VARSTORE_EFI with a specified VarStoreId must appear in the IFR before it can be referenced by a question.

33.3.8.3.92 EFI_IFR_VARSTOREDEVICE

Summary
Select the device which contains the variable store.

Prototype

```c
#define EFI_IFR_VARSTOREDEVICE_OP 0x27
typedef struct _EFI_IFR_VARSTOREDEVICE {
    EFI_IFR_OP_HEADER Header;
    EFI_STRING_ID DevicePath;
} EFI_IFR_VARSTOREDEVICE;
```

Members

- **Header**
  The byte sequence that defines the type of opcode as well as the length of the opcode being defined. For this tag, `Header.OpCode = EFI_IFR_VARSTOREDEVICE_OP`.

- **DevicePath**
  Specifies the string which contains the device path of the device where the variable store resides.

Description

This opcode describes the device path where a variable store resides. Normally, the Forms Processor finds the variable store on the handle specified when the HII database function NewPackageList() was called. However, if this opcode is found in the scope of a question, the handle specified by the text device path DevicePath is used instead.

33.3.8.3.93 EFI_IFR_VERSION

Summary
Push the version of the UEFI specification to which the Forms Processor conforms.

Prototype

```c
#define EFI_IFR_VERSION_OP 0x28
typedef struct _EFI_IFR_VERSION {
    EFI_IFR_OP_HEADER Header;
} EFI_IFR_VERSION;
```

Members

- **Header**
  The sequence that defines the type of opcode as well as the length of the opcode being defined. For this tag, `Header.OpCode = EFI_IFR_VERSION_OP`. 
Description

Returns the revision level of the UEFI specification with which the Forms Processor is compliant as a 16-bit unsigned integer, with the form:

[15:8]Major revision
[7:4]Tens digit of the minor revision
[3:0]Ones digit of the minor revision

The fields of the version have the following correlation with the revision of the UEFI system table.

- **Major revision**: \( \text{EFI.SYSTEM_TABLE_REVISION} \gg 16 \)
- **Tens digit of the minor revision**: \( (\text{EFI.SYSTEM_TABLE_REVISION} \& 0xFFFF)/10 \)
- **Ones digit of the minor revision**: \( (\text{EFI.SYSTEM_TABLE_REVISION} \& 0xFFFF)\%10 \)

### 33.3.8.3.94 EFI_IFR_WRITE

**Summary**

Change a value for the current question.

**Prototype**

```c
#define EFI_IFR_WRITE_OP 0x2E

typedef struct _EFI_IFR_WRITE {
    EFI_IFR_OP_HEADER Header;
} EFI_IFR_WRITE;
```

**Parameters**

**Header**

The sequence that defines the type of opcode as well as the length of the opcode being defined. For this tag, `Header.OpCode = EFI_IFR_WRITE_OP`

**Description**

Before writing the value of the current question to storage (if any storage was specified), the `this` constant is set (see `EFI_IFR_THIS`) and then this expression is evaluated.

### 33.3.8.3.95 EFI_IFR_ZERO

**Summary**

Push a zero on to the expression stack.
Prototype

```c
#define EFI_IFR_ZERO_OP 0x52
typedef struct _EFI_IFR_ZERO {
    EFI_IFR_OP_HEADER Header;
} EFI_IFR_ZERO;
```

Members

- **Header**: The sequence that defines the type of opcode as well as the length of the opcode being defined. For this tag, `Header.OpCode = EFI_IFR_ZERO_OP`.

Description

Push a zero on to the expression stack.

### 33.3.8.3.96 EFI_IFR_WARNING_IF

**Summary**

Creates a validation expression and warning message for a question.

Prototype

```c
#define EFI_IFR_WARNING_IF_OP 0x063
typedef struct _EFI_IFR_WARNING_IF {
    EFI_IFR_OP_HEADER Header;
    EFI_STRING_ID Warning;
    UINT8 TimeOut;
} EFI_IFR_WARNING_IF;
```

Members

- **Header**: The byte sequence that defines the type of opcode as well as the length of the opcode being defined. `Header.OpCode = EFI_IFR_WARNING_IF_OP`.
- **Warning**: The string token reference to the string that will be used for the warning check message.
- **TimeOut**: The number of seconds for the warning message to be displayed before it is timed out or acknowledged by the user. A value of zero indicates that the message is displayed indefinitely until the user acknowledges it.

Description

This tag uses a Boolean expression to allow the IFR creator to check options in a question, and provide a warning message if the expression is true. For example, this tag might be used to give a warning if the user attempts to disable a security setting, or change the value of a sensitive question. The tag provides a string to be used in a warning display to alert the user of the consequences of changing the question value. Warning tags will be evaluated when the user traverses from tag to tag. The browser must display the warning text message and not allow the form to be submitted until either the user acknowledges the message (with some key press for instance) or the number of seconds in TimeOut elapses. Unlike
inconsistency tags, the user should still be allowed to submit the results of a form even if the warning expression evaluates to true.

### 33.3.8.3.97 EFI_IFR_MATCH2

**Summary**

Pop a source string and a pattern string, push **TRUE** if the source string matches the Regular Expression pattern specified by the pattern string, otherwise push **FALSE**.

**Prototype**

```c
#define EFI_IFR_MATCH2_OP         0x64
typedef struct _EFI_IFR_MATCH2 {
    EFI_IFR_OP_HEADER           Header;
    EFI_GUID                    SyntaxType;
} EFI_IFR_MATCH2;
```

**Members**

- **Header**: Standard opcode header, where `Header.Opcode` is `EFI_IFR_MATCH2_OP`.
- **SyntaxType**: A GUID that identifies the regular expression syntax type to use for the **pattern** string. See `EFI_REGULAR_EXPRESSION_PROTOCOL` for current syntax definitions.

**Description**

This opcode performs the following actions:

1. Pop two values from the expression stack. The first value popped is the **string** and the second value popped is the **pattern**.
2. If the **string** or the **pattern** cannot be evaluated as a string, then push Undefined.
3. Call `GetInfo` function of each instance of `EFI_REGULAR_EXPRESSION_PROTOCOL`, looking for a `SyntaxType` that is listed in the set of supported regular expression syntax types returned by `RegExSyntaxTypeList`. If the type specified by `SyntaxType` is not supported in any of the `EFI_REGULAR_EXPRESSION_PROTOCOL` instances, or no `EFI_REGULAR_EXPRESSION_PROTOCOL` instance was found, push Undefined.
4. Process the **string** and **pattern** using the `MatchString` function of the `EFI_REGULAR_EXPRESSION_PROTOCOL` instance that supports the `SyntaxType`, where `SyntaxType` is the `SyntaxType` input to `MatchString`.
5. If the returned regular expression `Result` is **TRUE**, then push **TRUE**.
6. If the return regular expression `Result` is **FALSE**, then push **FALSE**.

**Note:** To ensure interoperability, drivers that publish HII IFR Forms packages should check the system capabilities by calling the `GetInfo` function of each `EFI_REGULAR_EXPRESSION_PROTOCOL` instance during initialization. If the required regular expression syntax type is not supported, the driver may install its own instance of `EFI_REGULAR_EXPRESSION_PROTOCOL` to add the support.
The driver may also choose to fall back to other methods of validation, such as using EFI_IFR_MATCH or callbacks.

33.3.9 Keyboard Package

```c
//***************************************************
// EFI_HII_KEYBOARD_PACKAGE_HDR
//***************************************************
typedef struct {
  EFI_HII_PACKAGE_HDR   Header;
  UINT16                LayoutCount;
  //EFI_HII_KEYBOARD_LAYOUT  Layout[];
} EFI_HII_KEYBOARD_PACKAGE_HDR;
```

- **Header**: The general pack header which defines both the type of pack and the length of the entire pack.
- **LayoutCount**: The number of keyboard layouts contained in the entire keyboard pack.
- **Layout**: An array of `LayoutCount` number of keyboard layouts.

33.3.10 Animations Package

The Animation package record describes how, when, and which EFI images to display. The package consists of two parts: a fixed header and the animation information.

33.3.10.1 Animated Images Package

**Summary**

The fixed header consists of a standard record header and the

**Prototype**

```c
typedef struct _EFI_HII_ANIMATION_PACKAGE_HDR {
  EFI_HII_ANIMATION_PACKAGE            Header;
  UINT32                                AnimationInfoOffset;
} EFI_HII_ANIMATION_PACKAGE_HDR;
```

- **Header**: Standard image header, where `Header.BlockType = EFI_HII_PACKAGE_ANIMATIONS`.
- **AnimationInfoOffset**: Offset, relative to this header, of the animation information. If this is zero, then there are no animation sequences in the package.

33.3.10.2 Animation Information

For each animated image identifier, the animation information gives a sequence of EFI images to display and how and when to transition to the next image. The animation information is encoded as a series of
blocks, with each block prefixed by a single byte header (**EFI_HII_ANIMATION_BLOCK** or one of the extension headers (**EFI_HII_AIBT_EXTx_BLOCK**). The blocks must be processed in order.

![Animation Information Encoded in Blocks](image)

**Figure 33-50 Animation Information Encoded in Blocks**

**Prototype**

```c
typedef struct _EFI_HII_ANIMATION_BLOCK {
    UINT8 BlockType;
    //UINT8 BlockBody[];
} EFI_HII_ANIMATION_BLOCK;
```

The following table describes the different block types:

**Table 33-12 Animation Block Types**

<table>
<thead>
<tr>
<th>Name</th>
<th>Value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>EFI_HII_AIBT_END</td>
<td>0x00</td>
<td>The end of the animation information.</td>
</tr>
<tr>
<td>EFI_HII_AIBT_OVERLAY_IMAGES</td>
<td>0x10</td>
<td>Animate sequence once by displaying the next image in the logical window.</td>
</tr>
<tr>
<td>EFI_HII_AIBT_CLEAR_IMAGES</td>
<td>0x11</td>
<td>Animate sequence once by clearing the logical window before displaying the next image.</td>
</tr>
</tbody>
</table>
In order to recreate all animation sequences, start at the first block and process them all until either an EFI_HII_AIBT_END block is found. When processing the animation blocks, each block refers to the current animation identifier (AnimationIdCurrent), which is initially set to one (1).

Animation blocks of an unknown type should be skipped. If they cannot be skipped, then processing halts.

### 33.3.10.2.1 EFI_HII_AIBT_END

**Summary**
Marks the end of the animation information.

**Prototype**
None

**Members**

<table>
<thead>
<tr>
<th>Name</th>
<th>Value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>EFI_HII_AIBT_DUPLICATE</td>
<td>0x20</td>
<td>Duplicate an existing animation identifier</td>
</tr>
<tr>
<td>EFI_HII_AIBT_SKIP2</td>
<td>0x21</td>
<td>Skip a certain number of animation identifiers.</td>
</tr>
<tr>
<td>EFI_HII_AIBT_SKIP1</td>
<td>0x22</td>
<td>Skip a certain number of animation identifiers.</td>
</tr>
<tr>
<td>EFI_HII_AIBT_EXT1</td>
<td>0x30</td>
<td>For future expansion (one byte length field)</td>
</tr>
<tr>
<td>EFI_HII_AIBT_EXT2</td>
<td>0x31</td>
<td>For future expansion (two byte length field)</td>
</tr>
<tr>
<td>EFI_HII_AIBT_EXT4</td>
<td>0x32</td>
<td>For future expansion (four byte length field)</td>
</tr>
</tbody>
</table>

EFI_HII_AIBT_RESTORE_SCRN 0x12 Animate sequence once by clearing the restoring the logical window before displaying the next image.

EFI_HII_AIBT_OVERLAY_IMAGES_LOOP 0x18 Animate repeating sequence by displaying the next image in the logical window.

EFI_HII_AIBT_CLEAR_IMAGES_LOOP 0x19 Animate repeating sequence by clearing the logical window before displaying the next image.

EFI_HII_AIBT_RESTORE_SCRN_LOOP 0x1A Animate repeating sequence by clearing the restoring the logical window before displaying the next image.

 EFI_HII_AIBT_DUPLICATE 0x20 Duplicate an existing animation identifier
 EFI_HII_AIBT_SKIP2 0x21 Skip a certain number of animation identifiers.
 EFI_HII_AIBT_SKIP1 0x22 Skip a certain number of animation identifiers.
 EFI_HII_AIBT_EXT1 0x30 For future expansion (one byte length field)
 EFI_HII_AIBT_EXT2 0x31 For future expansion (two byte length field)
 EFI_HII_AIBT_EXT4 0x32 For future expansion (four byte length field)

**Discussion**
Any animation sequences with an animation identifier greater than or equal to AnimationIdCurrent are empty. There is no additional data.

### 33.3.10.2.2 EFI_HII_AIBT_EXT1, EFI_HII_AIBT_EXT2, EFI_HII_AIBT_EXT4

**Summary**
Generic prefix for animation information with a 1-byte,2-byte or 4-byte length.
Prototype

```c
typedef struct _EFI_HII_AIBT_EXT1_BLOCK {
    EFI_HII_ANIMATION_BLOCK    Header;
    UINT8            BlockType2;
    UINT8            Length;
} EFI_HII_AIBT_EXT1_BLOCK;

typedef struct _EFI_HII_AIBT_EXT2_BLOCK {
    EFI_HII_ANIMATION_BLOCK    Header;
    UINT8            BlockType2;
    UINT16           Length;
} EFI_HII_AIBT_EXT2_BLOCK;

typedef struct _EFI_HII_AIBT_EXT4_BLOCK {
    EFI_HII_ANIMATION_BLOCK    Header;
    UINT8            BlockType2;
    UINT32           Length;
} EFI_HII_AIBT_EXT4_BLOCK;
```

Members

- **Header**: Standard animation header, where `Header.BlockType` = `EFI_HII_AIBT_EXT1`, `EFI_HII_AIBT_EXT2`, or `EFI_HII_AIBT_EXT4`.
- **Length**: Size of the animation block, in bytes, including the animation block header.
- **BlockType2**: The block type, as described in Table 33-9 on page 1825.

Discussion

These records are used for variable sized animation records which need an explicit length.

### 33.3.10.2.3 EFI_HII_AIBT_OVERLAY_IMAGES

**Summary**

An animation block to describe an animation sequence that does not cycle, and where one image is simply displayed over the previous image.
Prototype

```c
typedef struct _EFI_HII_AIBT_OVERLAY_IMAGES_BLOCK {
    EFI_IMAGE_ID          DftImageId;
    UINT16                Width;
    UINT16                Height;
    UINT16                CellCount;
    EFI_HII_ANIMATION_CELL AnimationCell[];
} EFI_HII_AIBT_OVERLAY_IMAGES_BLOCK;
```

Members

- **DftImageId**: This is the image that is to be referenced by the image protocols, if the animation function is not supported or disabled. This image can be one particular image from the animation sequence (if any one of the animation frames has a complete image) or an alternate image that can be displayed alone. If the value is zero, no image is displayed.

- **Width**: The overall width of the set of images (logical window width).

- **Height**: The overall height of the set of images (logical window height).

- **CellCount**: The number of `EFI_HII_ANIMATION_CELL` contained in the animation sequence.

- **AnimationCell**: An array of `CellCount` animation cells. The type `EFI_HII_ANIMATION_CELL` is defined in “Related Definitions” below.

Description

This record assigns the animation sequence data to the `AnimationIdCurrent` identifier and increments `AnimationIdCurrent` by one. This animation sequence is meant to be displayed only once (it is not a repeating sequence). Each image in the sequence will remain on the screen for the specified `delay` before the next image in the sequence is displayed.

The header type (either `BlockType` in `EFI_HII_ANIMATION_BLOCK` or `BlockType2` in `EFI_HII_AIBT_EXTx_BLOCK`) will be set to `EFI_HII_AIBT_OVERLAY_IMAGES`.

Related Definition

```c
typedef struct _EFI_HII_ANIMATION_CELL {
    UINT16          OffsetX;
    UINT16          OffsetY;
    EFI_IMAGE_ID    ImageId;
    UINT16          Delay;
} EFI_HII_ANIMATION_CELL;
```

- **OffsetX**: The X offset from the upper left hand corner of the logical window to position the indexed image.

- **OffsetY**: The Y offset from the upper left hand corner of the logical window to position the indexed image.
**ImageId**
The image to display at the specified offset from the upper left hand corner of the logical window.

**Delay**
The number of milliseconds to delay after displaying the indexed image and before continuing on to the next linked image. If value is zero, no delay.

**Related Description**
The logical window definition allows the animation to be centered, even though the first image might be way off center (bounds the sequence of images). All images will be clipped to the defined logical window, since the logical window is suppose to bound all images, normally there is nothing to clip. The DftImageId definition allows an alternate image to be displayed if animation is currently not supported. The DftImageId image is to be centered in the defined logical window.

### 33.3.10.2.4 EFI_HII_AIBT_CLEAR_IMAGES

**Summary**
An animation block to describe an animation sequence that does not cycle, and where the logical window is cleared to the specified color before the next image is displayed.

**Prototype**
```c
typedef struct _EFI_HII_AIBT_CLEAR_IMAGES_BLOCK {
  EFI_IMAGE_ID      DftImageId;
  UINT16            Width;
  UINT16            Height;
  UINT16            CellCount;
  EFI_HII_RGB_PIXEL BackgndColor;
  EFI_HII_ANIMATION_CELL AnimationCell[];
} EFI_HII_AIBT_CLEAR_IMAGES_BLOCK;
```

**Members**
- **DftImageId**
  This is image that is to be reference by the image protocols, if the animation function is not supported or disabled. This image can be one particular image from the animation sequence (if any one of the animation frames has a complete image) or an alternate image that can be displayed alone. If the value is zero, no image is displayed.

- **Width**
  The overall width of the set of images (logical window width).

- **Height**
  The overall height of the set of images (logical window height).

- **CellCount**
  The number of EFI_HII_ANIMATION_CELL contained in the animation sequence.
BackgndColor

The color to clear the logical window to before displaying the indexed image.

AnimationCell

An array of CellCount animation cells. The type EFI_HII_ANIMATION_CELL is defined in “Related Definitions” in EFI_HII_AIBT_OVERLAY_IMAGES.

Description

This record assigns the animation sequence data to the AnimationIdCurrent identifier and increment AnimationIdCurrent by one. This animation sequence is meant to be displayed only once (it is not a repeating sequence). Each image in the sequence will remain on the screen for the specified delay before the logical window is cleared to the specified color (BackgndColor) and the next image is displayed. The logical window is also cleared to the specified color before displaying the DftImageId image.

The header type (either BlockType in EFI_HII_ANIMATION_BLOCK or BlockType2 in EFI_HII_AIBT_EXTx_BLOCK) will be set to EFI_HII_AIBT_CLEAR_IMAGES.

33.3.10.2.5 EFI_HII_AIBT_RESTORE_SCRN

Summary

An animation block to describe an animation sequence that does not cycle, and where the screen is restored to the original state before the next image is displayed.

Prototype

typedef struct _EFI_HII_AIBT_RESTORE_SCRN_BLOCK {
    EFI_IMAGE_ID DftImageId;
    UINT16 Width;
    UINT16 Height;
    UINT16 CellCount;
    EFI_HII_ANIMATION_CELL AnimationCell[];
} EFI_HII_AIBT_RESTORE_SCRN_BLOCK;

Members

DftImageId

This is image that is to be reference by the image protocols, if the animation function is not supported or disabled. This image can be one particular image from the animation sequence (if any one of the animation frames has a complete image) or an alternate image that can be displayed alone. If the value is zero, no image is displayed.

Width

The overall width of the set of images (logical window width).

Height

The overall height of the set of images (logical window height).

CellCount

The number of EFI_HII_ANIMATION_CELL contained in the animation sequence.

AnimationCell

An array of CellCount animation cells. The type EFI_HII_ANIMATION_CELL is defined in “Related Definitions” in EFI_HII_AIBT_OVERLAY_IMAGES.
Description
This record assigns the animation sequence data to the AnimationIdCurrent identifier and increment AnimationIdCurrent by one. This animation sequence is meant to be displayed only once (it is not a repeating sequence). Before the first image is displayed, the entire defined logical window is saved to a buffer. Then each image in the sequence will remain on the screen for the specified delay before the logical window is restored to the original state and the next image is displayed.

If memory buffers are not available to save the logical window, this structure is treated as EFI_HII_AIBT_CLEAR IMAGES structure, with the BackgndColor value set to black.

The header type (either BlockType in EFI_HII_ANIMATION_BLOCK or BlockType2 in EFI_HII_AIBT_EXTx_BLOCK) will be set to EFI_HII_AIBT_RESTORE_SCRN.

33.3.10.2.6 EFI_HII_AIBT_OVERLAY_IMAGES_LOOP

Summary
An animation block to describe an animation sequence that continuously cycles, and where one image is simply displayed over the previous image.

Prototype
typedef EFI_HII_AIBT_OVERLAY_IMAGES_BLOCK
EFI_HII_AIBT_OVERLAY_IMAGES_LOOP_BLOCK {
    EFI_IMAGE_ID            DftImageId;
    UINT16                  Width;
    UINT16                  Height;
    UINT16                  CellCount;
    EFI_HII_ANIMATION_CELL  AnimationCell[];
} EFI_HII_AIBT_OVERLAY_IMAGES_LOOP_BLOCK;

Members
DftImageId          This is image that is to be reference by the image protocols, if the animation function is not supported or disabled. This image can be one particular image from the animation sequence (if any one of the animation frames has a complete image) or an alternate image that can be displayed alone. If the value is zero, no image is displayed.
Width                The overall width of the set of images (logical window width).
Height               The overall height of the set of images (logical window height).
CellCount            The number of EFI_HII_ANIMATION_CELL contained in the animation sequence.
AnimationCell        An array of CellCount animation cells. The type EFI_HII_ANIMATION_CELL is defined in “Related Definitions” in EFI_HII_AIBT_OVERLAY_IMAGES
Description
This record assigns the animation sequence data to the AnimationIdCurrent identifier and increment AnimationIdCurrent by one. This animation sequence is meant to continuously cycle until stopped or paused. Each image in the sequence will remain on the screen for the specified delay before the next image in the sequence is displayed.

The header type (either BlockType in EFI_HII_ANIMATION_BLOCK or BlockType2 in EFI_HII_AIBT_EXTx_BLOCK) will be set to EFI_HII_AIBT_OVERLAY_IMAGES_LOOP.

33.3.10.2.7 EFI_HII_AIBT_CLEAR_IMAGES_LOOP

Summary
An animation block to describe an animation sequence that continuously cycles, and where the logical window is cleared to the specified color before the next image is displayed.

Prototype

typedef EFI_HII_AIBT_CLEAR_IMAGES_BLOCK EFI_HII_AIBT_CLEAR_IMAGES_LOOP_BLOCK
{
    EFI_IMAGE_ID    DftImageId;
    UINT16          Width;
    UINT16          Height;
    UINT16          CellCount;
    EFI_HII_RGB_PIXEL BackgndColor;
    EFI_HII_ANIMATION_CELL AnimationCell[];
} EFI_HII_AIBT_CLEAR_IMAGES_LOOP_BLOCK;

Members

DftImageId This is image that is to be reference by the image protocols, if the animation function is not supported or disabled. This image can be one particular image from the animation sequence (if any one of the animation frames has a complete image) or an alternate image that can be displayed alone. If the value is zero, no image is displayed.

Width The overall width of the set of images (logical window width).
Height The overall height of the set of images (logical window height).

CellCount The number of EFI_HII_ANIMATION_CELL contained in the animation sequence.

BackgndColor The color to clear the logical window to before displaying the indexed image.

AnimationCell An array of CellCount animation cells. The type EFI_HII_ANIMATION_CELL is defined in “Related Definitions” in EFI_HII_AIBT_OVERLAY_IMAGES
Description
This record assigns the animation sequence data to the `AnimationIdCurrent` identifier and increment `AnimationIdCurrent` by one. This animation sequence is meant to continuously cycle until stopped or paused. Each image in the sequence will remain on the screen for the specified `delay` before the logical window is cleared to the specified color (`BackgndColor`) and the next image is displayed. The logical window is also cleared to the specified color before displaying the `DftImageId` image.

The header type (either `BlockType` in `EFI_HII_ANIMATION_BLOCK` or `BlockType2` in `EFI_HII_AIBT_EXTx_BLOCK`) will be set to `EFI_HII_AIBT_CLEAR_IMAGES_LOOP`.

33.3.10.2.8 EFI_HII_AIBT_RESTORE_SCRN_LOOP

Summary
An animation block to describe an animation sequence that continuously cycles, and where the screen is restored to the original state before the next image is displayed.

Prototype
```c
typedef EFI_HII_AIBT_RESTORE_SCRN_LOOP_BLOCK
    EFI_HII_AIBT_RESTORE_SCRN_LOOP_BLOCK {
    EFI_IMAGE_ID       DftImageId;
    UINT16             Width;
    UINT16             Height;
    UINT16             CellCount;
    EFI_HII_ANIMATION_CELL AnimationCell[];
} EFI_HII_AIBT_RESTORE_SCRN_LOOP_BLOCK;
```

Members
- **Header** Standard image header, where `Header.BlockType = EFI_HII_AIBT_RESTORE_SCRN_LOOP`.
- **DftImageId** This is image that is to be reference by the image protocols, if the animation function is not supported or disabled. This image can be one particular image from the animation sequence (if any one of the animation frames has a complete image) or an alternate image that can be displayed alone. If the value is zero, no image is displayed.
- **Length** Size of the animation block, in bytes, including the animation block header.
- **Width** The overall width of the set of images (logical window width).
- **Height** The overall height of the set of images (logical window height).
- **CellCount** The number of `EFI_HII_ANIMATION_CELL` contained in the animation sequence.
- **AnimationCell** An array of `CellCount` animation cells. The type `EFI_HII_ANIMATION_CELL` is defined in “Related Definitions” in `EFI_HII_AIBT_OVERLAY_IMAGES`
Description
This record assigns the animation sequence data to the AnimationIdCurrent identifier and increment AnimationIdCurrent by one. This animation sequence is meant to continuously cycle until stopped or paused. Before the first image is displayed, the entire defined logical window is saved to a buffer. Then each image in the sequence will remain on the screen for the specified delay before the logical window is restored to the original state and the next image is displayed.

If memory buffers are not available to save the logical window, this structure is treated as EFI_HII_AIBT_CLEAR_IMAGES_LOOP structure, with the BackgndColor value set to black.

The header type (either BlockType in EFI_HII_ANIMATION_BLOCK or BlockType2 in EFI_HII_AIBT_EXTx_BLOCK) will be set to EFI_HII_AIBT_RESTORE_SCRN_LOOP.

33.3.10.2.9 EFI_HII_AIBT_DUPLICATE

Summary
Assigns a new character value to a previously defined animation sequence.

Prototype
```c
typedef struct _EFI_HII_AIBT_DUPLICATE_BLOCK {
    EFI_ANIMATION_ID AnimationId;
} EFI_HII_AIBT_DUPLICATE_BLOCK;
```

Members
- **AnimationId** The previously defined animation ID with the exact same animation information.

Discussion
Indicates that the animation sequence with animation ID AnimationIdCurrent has the same animation information as a previously defined animation ID and increments AnimationIdCurrent by one.

The header type (either BlockType in EFI_HII_ANIMATION_BLOCK or BlockType2 in EFI_HII_AIBT_EXTx_BLOCK) will be set to EFI_HII_AIBT_DUPLICATE.

33.3.10.2.10 EFI_HII_AIBT_SKIP1

Summary
Skips animation IDs.

Prototype
```c
typedef struct _EFI_HII_AIBT_SKIP1_BLOCK {
    UINT8 SkipCount;
} EFI_HII_AIBT_SKIP1_BLOCK;
```

Members
- **SkipCount** The unsigned 8-bit value to add to AnimationIdCurrent.
Discussion
Increments the current animation ID `AnimationIdCurrent` by the number specified. The header type (either `BlockType` in `EFI_HII_ANIMATION_BLOCK` or `BlockType2` in `EFI_HII_AIBT_EXTx_BLOCK`) will be set to `EFI_HII_AIBT_SKIP1`.

33.3.10.2.11 EFI_HII_AIBT_SKIP2

Summary
Skips animation IDs.

Prototype
```c
typedef struct _EFI_HII_AIBT_SKIP2_BLOCK {
    UINT16 SkipCount;
} EFI_HII_AIBT_SKIP2_BLOCK;
```

Members
- `SkipCount` The unsigned 16-bit value to add to `AnimationIdCurrent`.

Discussion
Increments the current animation ID `AnimationIdCurrent` by the number specified. The header type (either `BlockType` in `EFI_HII_ANIMATION_BLOCK` or `BlockType2` in `EFI_HII_AIBT_EXTx_BLOCK`) will be set to `EFI_HII_AIBT_SKIP2`. 
34 - HII Protocols

This section provides code definitions for the HII-related protocols, functions, and type definitions, which are the required architectural mechanisms by which UEFI-compliant systems manage user input. The major areas described include the following:

- Font management.
- String management.
- Image management.
- Database management.

34.1 Font Protocol

 EFI_HII_FONT_PROTOCOL

Summary

Interfaces which retrieve font information.

GUID

#define EFI_HII_FONT_PROTOCOL_GUID \
{ 0xe9ca4775, 0x8657, 0x47fc, \ 
{0x97, 0xe7, 0x7e, 0xd6, 0x5a, 0x8, 0x43, 0x24 }}

Protocol

typedef struct _EFI_HII_FONT_PROTOCOL {
  EFI_HII_STRING_TO_IMAGE StringToImage;
  EFI_HII_STRING_ID_TO_IMAGE StringIdToImage;
  EFI_HII_GET_GLYPH Glyph;
  EFI_HII_GET_FONT_INFO GetFontInfo;
} EFI_HII_FONT_PROTOCOL;

Members

StringToImage, StringIdToImage
  Render a string to a bitmap or to the display.
Glyph
  Return a specific glyph in a specific font.
GetFontInfo
  Return font information for a specific font.

 EFI_HII_FONT_PROTOCOL.StringToImage()  

Summary

Renders a string to a bitmap or to the display.
Prototype

typedef

EFI_STATUS

(EIFIAPI *EFI_HII_STRING_TO_IMAGE) (
    IN CONST EFI_HII_FONT_PROTOCOL *This,
    IN EFI_HII_OUT_FLAGS Flags,
    IN CONST EFI_STRING String,
    IN CONST EFI_FONT_DISPLAY_INFO *StringInfo OPTIONAL,
    IN OUT EFI_IMAGE_OUTPUT **Blt,
    IN UINTN BltX,
    IN UINTN BltY,
    OUT EFI_HII_ROW_INFO **RowInfoArray OPTIONAL,
    OUT UINTN *RowInfoArraySize OPTIONAL,
    OUT UINTN *ColumnInfoArray OPTIONAL
);

Parameters

This

A pointer to the EFI_HII_FONT_PROTOCOL instance.

Flags

Describes how the string is to be drawn. EFI_HII_OUT_FLAGS is defined in Related Definitions, below.

String

Points to the null-terminated string to be displayed.

StringInfo

Points to the string output information, including the color and font. If NULL, then the string will be output in the default system font and color.

Blt

If this points to a non-NULL on entry, this points to the image, which is Blt.Width pixels wide and Blt.Height pixels high. The string will be drawn onto this image and EFI_HII_OUT_FLAG_CLIP is implied. If this points to a NULL on entry, then a buffer will be allocated to hold the generated image and the pointer updated on exit. It is the caller’s responsibility to free this buffer.

BltX, BltY

Specifies the offset from the left and top edge of the image of the first character cell in the image.

RowInfoArray

If this is non-NULL on entry, then on exit, this will point to an allocated buffer containing row information and RowInfoArraySize will be updated to contain the number of elements. This array describes the characters which were at least partially drawn and the heights of the rows. It is the caller’s responsibility to free this buffer.
**RowInfoArraySize**

If this is non-NULL on entry, then on exit it contains the number of elements in `RowInfoArray`.

**ColumnInfoArray**

If this is non-NULL, then on return it will be filled with the horizontal offset for each character in the string on the row where it is displayed. Non-printing characters will have the offset ~0. The caller is responsible to allocate a buffer large enough so that there is one entry for each character in the string, not including the null-terminator. It is possible when character display is normalized that some character cells overlap.

**Description**

This function renders a string to a bitmap or the screen using the specified font, color and options. It either draws the string and glyphs on an existing bitmap, allocates a new bitmap or uses the screen. The strings can be clipped or wrapped. Optionally, the function also returns the information about each row and the character position on that row.

If `EFI_HII_OUT_FLAG_CLIP` is set, then text will be formatted only based on explicit line breaks and all pixels which would lie outside the bounding box specified by `Blt.Width` and `Blt.Height` are ignored. The information in the `RowInfoArray` only describes characters which are at least partially displayed. For the final row, the `RowInfoArray.LineHeight` and `RowInfoArray.BaseLine` may describe pixels which are outside the limit specified by `Blt.Height` (unless `EFI_HII_OUT_FLAG_CLIP_CLEAN_Y` is specified) even though those pixels were not drawn. The `LineWidth` may describe pixels which are outside the limit specified by `Blt.Width` (unless `EFI_HII_OUT_FLAG_CLIP_CLEAN_X` is specified) even though those pixels were not drawn.

If `EFI_HII_OUT_FLAG_CLIP_CLEAN_X` is set, then it modifies the behavior of `EFI_HII_OUT_FLAG_CLIP` so that if a character's right-most pixel cannot fit, then it will not be drawn at all. This flag requires that `EFI_HII_OUT_FLAG_CLIP` be set.

If `EFI_HII_OUT_FLAG_CLIP_CLEAN_Y` is set, then it modifies the behavior of `EFI_HII_OUT_FLAG_CLIP` so that if a row's bottom-most pixel cannot fit, then it will not be drawn at all. This flag requires that `EFI_HII_OUT_FLAG_CLIP` be set.

If `EFI_HII_OUT_FLAG_WRAP` is set, then text will be wrapped at the right-most line-break opportunity prior to a character whose right-most extent would exceed `Blt.Width`. If no line-break opportunity can be found, then the text will behave as if `EFI_HII_OUT_FLAG_CLIP_CLEAN_X` is set. This flag cannot be used with `EFI_HII_OUT_FLAG_CLIP_CLEAN_X`.

If `EFI_HII_OUT_FLAG_TRANSPARENT` is set, then `StringInfo.BackgroundColor` is ignored and all “off” pixels in the character’s drawn will use the pixel value from `Blt`. This flag cannot be used if `Blt` is NULL upon entry.

If `EFI_HII_IGNORE_IF_NO_GLYPH` is set, then characters which have no glyphs are not drawn. Otherwise, they are replaced with Unicode character code 0xFFFD (REPLACEMENT CHARACTER).

If `EFI_HII_IGNORE_LINE_BREAK` is set, then explicit line break characters will be ignored.

If `EFI_HII_DIRECT_TO_SCREEN` is set, then the string will be written directly to the output device specified by `Screen`. Otherwise the string will be rendered to the bitmap specified by `Bitmap`. 
Related Definitions

typedef UINT32 EFI_HII_OUT_FLAGS;

#define EFI_HII_OUT_FLAG_CLIP     0x00000001
#define EFI_HII_OUT_FLAG_WRAP     0x00000002
#define EFI_HII_OUT_FLAG_CLIP_CLEAN_Y 0x00000004
#define EFI_HII_OUT_FLAG_CLIP_CLEAN_X 0x00000008
#define EFI_HII_OUT_FLAG_TRANSPARENT  0x00000010
#define EFI_HII_IGNORE_IF_NO_GLYPH   0x00000020
#define EFI_HII_IGNORE_LINE_BREAK   0x00000040
#define EFI_HII_DIRECT_TO_SCREEN    0x00000080

typedef CHAR16 *EFI_STRING;

typedef struct _EFI_HII_ROW_INFO {
    UINTN StartIndex;
    UINTN EndIndex;
    UINTN LineHeight;
    UINTN LineWidth;
    UINTN BaselineOffset;
} EFI_HII_ROW_INFO;

StartIndex
The index of the first character in the string which is displayed on the line.

EndIndex
The index of the last character in the string which is displayed on the line.

LineHeight
The height of the line, in pixels.

LineWidth
The width of the text on the line, in pixels.

BaselineOffset
The font baseline offset in pixels from the bottom of the row, or 0 if none.

Status Codes Returned

EFI_SUCCESS The string was successfully updated.
EFI_OUT_OF_RESOURCES Unable to allocate an output buffer for RowInfoArray or Blt.
EFI_INVALID_PARAMETER The String or Blt was NULL.
EFI_INVALID_PARAMETER Flags were invalid combination
 EFI_HII_FONT_PROTOCOL.StringIdToImage()

Summary
Render a string to a bitmap or the screen containing the contents of the specified string.

Prototype

```c
typedef
  EFI_STATUS
  (EFIAPI *EFI_HII_STRING_ID_TO_IMAGE) (  
    IN   CONST EFI_HII_FONT_PROTOCOL  *This,
    IN   EFI_HII_OUT_FLAGS          Flags,
    IN   EFI_HII_HANDLE             PackageList,
    IN   EFI_STRING_ID              StringId,
    IN   CONST CHAR8*               Language,
    IN   CONST EFI_FONT_DISPLAY_INFO *StringInfo OPTIONAL,
    IN   EFI_IMAGE_OUTPUT **Blt,
    IN   UINTN                      BltX,
    IN   UINTN                      BltY,
    OUT  EFI_HII_ROW_INFO **RowInfoArray OPTIONAL,
    OUT  UINTN                      *RowInfoArraySize OPTIONAL,
    OUT  UINTN                      *ColumnInfoArray OPTIONAL
  );
```

Parameters

- **This**
  A pointer to the EFI_HII_FONT_PROTOCOL instance.

- **Flags**
  Describes how the string is to be drawn. EFI_HII_OUT_FLAGS is defined in Related Definitions, below.

- **PackageList**
  The package list in the HII database to search for the specified string.

- **StringId**
  The string’s id, which is unique within PackageList.

- **Language**
  Points to the language for the retrieved string. If NULL, then the current system language is used.

- **StringInfo**
  Points to the string output information, including the color and font. If NULL, then the string will be output in the default system font and color.
Blt
If this points to a non-NULL on entry, this points to the image, which is `Blt.Width` pixels wide and `Height` pixels high. The string will be drawn onto this image and `EFI_HII_OUT_FLAG_CLIP` is implied. If this points to a NULL on entry, then a buffer will be allocated to hold the generated image and the pointer updated on exit. It is the caller’s responsibility to free this buffer.

BltX, BltY
Specifies the offset from the left and top edge of the output image of the first character cell in the image.

RowInfoArray
If this is non-NULL on entry, then on exit, this will point to an allocated buffer containing row information and `RowInfoArraySize` will be updated to contain the number of elements. This array describes the characters which were at least partially drawn and the heights of the rows. It is the caller’s responsibility to free this buffer.

RowInfoArraySize
If this is non-NULL on entry, then on exit it contains the number of elements in `RowInfoArray`.

ColumnInfoArray
If non-NULL, on return it is filled with the horizontal offset for each character in the string on the row where it is displayed. Non-printing characters will have the offset \textasciitilde0. The caller is responsible to allocate a buffer large enough so that there is one entry for each character in the string, not including the null-terminator. It is possible when character display is normalized that some character cells overlap.

Description
This function renders a string as a bitmap or to the screen and can clip or wrap the string. The bitmap is either supplied by the caller or else is allocated by the function. The strings are drawn with the font, size and style specified and can be drawn transparently oropaquely. The function can also return information about each row and each character’s position on the row.

If `EFI_HII_OUT_FLAG_CLIP` is set, then text will be formatted only based on explicit line breaks and all pixels which would lie outside the bounding box specified by `Width` and `Height` are ignored. The information in the `RowInfoArray` only describes characters which are at least partially displayed. For the final row, the LineHeight and BaseLine may describe pixels which are outside the limit specified by `Height` (unless `EFI_HII_OUT_FLAG_CLIP_CLEAN_Y` is specified) even though those pixels were not drawn.

If `EFI_HII_OUT_FLAG_CLIP_CLEAN_X` is set, then it modifies the behavior of `EFI_HII_OUT_FLAG_CLIP` so that if a character’s right-most on pixel cannot fit, then it will not be drawn at all. This flag requires that `EFI_HII_OUT_FLAG_CLIP` be set.

If `EFI_HII_OUT_FLAG_CLIP_CLEAN_Y` is set, then it modifies the behavior of `EFI_HII_OUT_FLAG_CLIP` so that if a row’s bottom most pixel cannot fit, then it will not be drawn at all. This flag requires that `EFI_HII_OUT_FLAG_CLIP` be set.
If `EFI_HII_OUT_FLAG_WRAP` is set, then text will be wrapped at the right-most line-break opportunity prior to a character whose right-most extent would exceed `Width`. If no line-break opportunity can be found, then the text will behave as if `EFI_HII_OUT_FLAG_CLIP_CLEAN_X` is set. This flag cannot be used with `EFI_HII_OUT_FLAG_CLIP_CLEAN_X`.

If `EFI_HII_OUT_FLAG_TRANSPARENT` is set, then `BackgroundColor` is ignored and all “off” pixels in the character’s glyph will use the pixel value from `Blt`. This flag cannot be used if `Blt` is NULL upon entry.

If `EFI_HII_IGNORE_IF_NO_GLYPH` is set, then characters which have no glyphs are not drawn. Otherwise, they are replaced with Unicode character code 0xFFFD (REPLACEMENT CHARACTER).

If `EFI_HII_IGNORE_LINE_BREAK` is set, then explicit line break characters will be ignored.

If `EFI_HII_DIRECT_TO_SCREEN` is set, then the string will be written directly to the output device specified by `Screen`. Otherwise the string will be rendered to the bitmap specified by `Bitmap`.

**Status Codes Returned**

- `EFI_SUCCESS` The string was successfully updated.
- `EFI_OUT_OF_RESOURCES` Unable to allocate an output buffer for `RowInfoArray` or `Blt`.
- `EFI_INVALID_PARAMETER` The `StringId` or `PackageList` was NULL.
- `EFI_INVALID_PARAMETER` Flags were invalid combination.
- `EFI_NOT_FOUND` The `StringId` is not in the specified `PackageList`.

**EFI_HII_FONT_PROTOCOL.GetGlyph()**

**Summary**

Return image and information about a single glyph.

**Prototype**

```c
typedef EFI_STATUS
  (EFIAPI *EFI_HII_GET_GLYPH) (
    IN CONST EFI_HII_FONT_PROTOCOL *This,
    IN CHAR16 Char,
    IN CONST EFI_FONT_DISPLAY_INFO *StringInfo,
    OUT EFI_IMAGE_OUTPUT **Blt;
    OUT UINTN *Baseline OPTIONAL;
  );
```

**Parameters**

- **This**
  A pointer to the `EFI_HII_FONT_PROTOCOL` instance.

- **Char**
  Character to retrieve.
**StringInfo**

Points to the string font and color information or NULL if the string should use the default system font and color.

**Blt**

Thus must point to a NULL on entry. A buffer will be allocated to hold the output and the pointer updated on exit. It is the caller’s responsibility to free this buffer. On return, only Blt.Width, Blt.Height, and Blt.Image.Bitmap are valid.

**Baseline**

Number of pixels from the bottom of the bitmap to the baseline.

**Description**

Convert the glyph for a single character into a bitmap.

**Status Codes Returned**

- EFI_SUCCESS: Glyph bitmap created.
- EFI_OUT_OF_RESOURCES: Unable to allocate the output buffer Blt.
- EFI_WARN_UNKNOWN_GLYPH: The glyph was unknown and was replaced with the glyph for Unicode character code 0xFFFD.
- EFI_INVALID_PARAMETER: Blt is NULL or *Blt is !Null

**EFI_HII_FONT_PROTOCOL.GetFontInfo()**

**Summary**

Return information about a particular font.

**Prototype**

```c
typedef EFI_STATUS (EFIAPI *EFI_HII_GET_FONT_INFO)(
    IN CONST EFI_HII_FONT_PROTOCOL *This,
    IN OUT EFI_FONT_HANDLE *FontHandle,
    IN CONST EFI_FONT_DISPLAY_INFO *StringInfoIn, OPTIONAL
    OUT EFI_FONT_DISPLAY_INFO **StringInfoOut,
    IN CONST EFI_STRING String OPTIONAL
);

typedef VOID *EFI_FONT_HANDLE;
```

**Parameters**

- **This**
  A pointer to the **EFI_HII_FONT_PROTOCOL** instance.
FontHandle

On entry, points to the font handle returned by a previous call to `GetFontInfo()` or points to NULL to start with the first font. On return, points to the returned font handle or points to NULL if there are no more matching fonts.

StringInfoIn

Upon entry, points to the font to return information about. If NULL, then the information about the system default font will be returned.

StringInfoOut

Upon return, contains the matching font’s information. If NULL, then no information is returned. This buffer is allocated with a call to the Boot Service `AllocatePool()`. It is the caller's responsibility to call the Boot Service `FreePool()` when the caller no longer requires the contents of StringInfoOut.

String

Points to the string which will be tested to determine if all characters are available. If NULL, then any font is acceptable.

Description

This function iterates through fonts which match the specified font, using the specified criteria. If String is non-NULL, then all of the characters in the string must exist in order for a candidate font to be returned.

Status Codes Returned

<table>
<thead>
<tr>
<th>Status Code</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>EFI_SUCCESS</td>
<td>Matching font returned successfully.</td>
</tr>
<tr>
<td>EFI_NOT_FOUND</td>
<td>No matching font was found.</td>
</tr>
<tr>
<td>EFI_OUT_OF_RESOURCES</td>
<td>There were insufficient resources to complete the request.</td>
</tr>
</tbody>
</table>

34.2 EFI HII Font Ex Protocol

The EFI HII Font Ex protocol defines an extension to the EFI HII Font protocol which enables various new capabilities described in this section.

EFI_HII_FONT_EX_PROTOCOL

Summary

Interfaces which retrieve the font information.
GUID

```
#define EFI_HII_FONT_EX_PROTOCOL_GUID \
{ 0x849e6875, 0xdb35, 0x4df8, 0xb4, \
{0x1e, 0xc8, 0xf3, 0x37, 0x18, 0x7, 0x3f }}
```

Protocol

```c
typedef struct _EFI_HII_FONT_EX_PROTOCOL {
  EFI_HII_STRING_TO_IMAGE_EX StringToImageEx;
  EFI_HII_STRING_ID_TO_IMAGE_EX StringIdToImageEx;
  EFI_HII_GET_GLYPH_EX GetGlyphEx;
  EFI_HII_GET_FONT_INFO_EX GetFontInfoEx;
  EFI_HII_GET_GLYPH_INFO GetGlyphInfo;
} EFI_HII_FONT_EX_PROTOCOL;
```

Members

- **StringToImageEx, StringIdToImageEx**
  Render a string to a bitmap or to the display. This function will try to use the external font glyph generator for generating the glyph if it can’t find the glyph in the font database.

- **GetGlyphEx**
  Return a specific glyph in a specific font. This function will try to use the external font glyph generator for generating the glyph if it can’t find the glyph in the font database.

- **GetFontInfoEx**
  Return the font information for a specific font, this protocol invokes original `EFI_HII_FONT_PROTOCOL.GetFontInfo()` implicitly.

- **GetGlyphInfo**
  Return the glyph information for the single glyph.

**EFI_HII_FONT_EX_PROTOCOL.StringToImageEx()**

Summary

Render a string to a bitmap or to the display. The prototype of this extension function is the same with `EFI_HII_FONT_PROTOCOL.StringToImage()`. 
Protocol

typedef
  EFI_STATUS
  (EFIAPI *EFI_HII_STRING_TO_IMAGE_EX)(
    IN CONST EFI_HII_FONT_EX_PROTOCOL *This,
    IN EFI_HII_OUT_FLAGS Flags,
    IN const EFI_STRING String,
    IN const EFI_FONT_DISPLAY_INFO *StringInfo OPTIONAL,
    IN OUT EFI_IMAGE_OUTPUT **Blt,
    IN UINTN BltX,
    IN UINTN BltY,
    OUT EFI_HII_ROW_INFO **RowInfoArray OPTIONAL,
    OUT UINTN *RowInfoArraySize OPTIONAL,
    OUT UINTN *ColumnInfoArray OPTIONAL
  );

Parameters
Same with EFI_HII_FONT_PROTOCOL.StringToImage().

Description
This function is similar to EFI_HII_FONT_PROTOCOL.StringToImage(). The difference is that this function will locate all EFI_HII_FONT_GLYPH_GENERATOR_PROTOCOL instances that are installed in the system when the glyph in the string with the given font information is not found in the current HII glyph database. The function will attempt to generate the glyph information and the bitmap using the first EFI_HII_FONT_GLYPH_GENERATOR_PROTOCOL instance that supports the requested font information in the EFI_FONT_DISPLAY_INFO.

Status Codes Returned
Same with EFI_HII_FONT_PROTOCOL.StringToImage().

EFI_HII_FONT_EX_PROTOCOL.StringIdToImageEx()

Summary
Render a string to a bitmap or the screen containing the contents of the specified string. The prototype of this extension function is the same with EFI_HII_FONT_PROTOCOL.StringIdToImage().
Protocol

typedef
EFI_STATUS
(EFIAPI *EFI_HII_STRING_ID_TO_IMAGE_EX)(
    IN CONST EFI_HII_FONT_EX_PROTOCOL *This,
    IN EFI_HII_OUT_FLAGS Flags,
    IN EFI_HII_HANDLE PackageList,
    IN EFI_STRING_ID StringId,
    IN CONST CHAR8 *Language,
    IN CONST EFI_FONT_DISPLAY_INFO *StringInfo OPTIONAL,
    IN OUT EFI_IMAGE_OUTPUT **Blt,
    IN UINTN BltX,
    IN UINTN BltY,
    OUT EFI_HII_ROW_INFO **RowInfoArray OPTIONAL,
    OUT UINTN *RowInfoArraySize OPTIONAL,
    OUT UINTN *ColumnInfoArray OPTIONAL,
);

Parameters

Same with EFI_HII_FONT_PROTOCOL.StringIdToImage().

Description

This function is similar to EFI_HII_FONT_PROTOCOL.StringIdToImage(). The difference is that this function will locate all EFI_HII_FONT_GLYPH_GENERATOR_PROTOCOL instances that are installed in the system when the glyph in the string with the given font information is not found in the current HII glyph database. The function will attempt to generate the glyph information and the bitmap using the first EFI_HII_FONT_GLYPH_GENERATOR_PROTOCOL instance that supports the requested font information in the EFI_FONT_DISPLAY_INFO.

Status Codes Returned

Same with EFI_HII_FONT_PROTOCOL.StringToImage().

EFI_HII_FONT_EX_PROTOCOL.GetGlyphEx()

Summary

Return image and baseline about a single glyph. The prototype of this extension function is the same with EFI_HII_FONT_PROTOCOL.GetGlyph().
Protocol

typedef

EFI_STATUS

EFI_HII_GET_GLYPH_EX(IN CONST EFI_HII_FONT_EX_PROTOCOL *This,
IN CHAR16 Char,
IN CONST EFI_FONT_DISPLAY_INFO *StringInfo,
IN OUT EFI_IMAGE_OUTPUT **Blt,
IN UINTN Baseline OPTIONAL);

Parameters

Same with EFI_HII_FONT_PROTOCOL.GetGlyph().

Description

This function is similar to EFI_HII_FONT_PROTOCOL.GetGlyph(). The difference is that this function will locate all EFI_HII_FONT_GLYPH_GENERATOR_PROTOCOL instances that are installed in the system when the glyph in the string with the given font information is not found in the current HII glyph database. The function will attempt to generate the glyph information and the bitmap using the first EFI_HII_FONT_GLYPH_GENERATOR_PROTOCOL instance that supports the requested font information in the EFI_FONT_DISPLAY_INFO.

Status Codes Returned

Same as EFI_HII_FONT_PROTOCOL.GetGlyph().

EFI_HII_FONT_EX_PROTOCOL.GetFontInfoEx()

Summary

Return information about a particular font. The prototype of this extension function is the same with EFI_HII_FONT_PROTOCOL.GetFontInformation().

Protocol

typedef

EFI_STATUS

EFI_HII_GET_FONT_INFO_EX(IN CONST EFI_HII_FONT_EX_PROTOCOL *This,
IN OUT EFI_FONT_HANDLE *FontHandle,
IN CONST EFI_FONT_DISPLAY_INFO *StringInfoIn, OPTIONAL
OUT EFI_FONT_DISPLAY_INFO **StringInfoOut,
IN CONST EFI_STRING String OPTIONAL)

Parameters
Same with `EFI_HII_FONT_PROTOCOL.GetFontInfo()`.

**Description**

Same with `EFI_HII_FONT_PROTOCOL.GetFontInfo()`. This protocol invokes `EFI_HII_FONT_PROTOCOL.GetFontInfo()` implicitly.

**Status Codes Returned**

Same as `EFI_HII_FONT_PROTOCOL.GetFontInfo()`.

### EFI_HII_FONT_EX_PROTOCOL.GetGlyphInfo()

**Summary**

The function returns the information of the single glyph.

**Protocol**

```c
typedef EFI_STATUS
   (EFIAPI *EFI_HII_GET_GLYPH_INFO)(
   IN CONST EFI_HII_FONT_EX_PROTOCOL *This,
   IN CHAR16 Char,
   IN CONST EFI_FONT_DISPLAY_INFO *FontDisplayInfo,
   OUT EFI_HII_GLYPH_INFO * GlyphInfo
   );
```

**Parameters**

- **This**: `EFI_HII_FONT_EX_PROTOCOL` instance.
- **Char**: Information of Character to retrieve.
- **FontDisplayInfo**: Font display information of this character.
- **GlyphInfo**: Pointer to retrieve the glyph information.

**Description**

This function returns the glyph information of character in the specific font family. This function will locate all `EFI_HII_FONT_GLYPH_GENERATOR` protocol instances that are installed in the system, and attempt to use them if it can’t find the glyph information in the font database. It returns `EFI_UNSUPPORTED` if neither the font database nor any instances of the `EFI_HII_FONT_GLYPH_GENERATOR` protocols support the font glyph in the specific font family. Otherwise, the `EFI_HII_GLYPH_INFO` is returned in `GlyphInfo`. This function only returns the glyph...
geometry information instead of allocating the buffer for `EFI_IMAGE_OUTPUT` and drawing the glyph in the buffer.

![Figure 34-1 Glyph Example](image)

**Status Codes Returned**

<table>
<thead>
<tr>
<th>Code Definition</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>EFI_SUCCESS</strong></td>
<td>The glyph information was returned to <code>GlyphInfo</code>.</td>
</tr>
<tr>
<td><strong>EFI_OUT_OF_RESOURCES</strong></td>
<td>Memory allocation failed in this function.</td>
</tr>
<tr>
<td><strong>EFI_NOT_FOUND</strong></td>
<td>The input character was not found in the database.</td>
</tr>
<tr>
<td><strong>EFI_UNSUPPORTED</strong></td>
<td>The font is not supported.</td>
</tr>
<tr>
<td><strong>EFI_INVALID_PARAMETER</strong></td>
<td>The <code>GlyphInfo</code> or <code>FontDisplayInfo</code> was NULL.</td>
</tr>
</tbody>
</table>

### 34.2.1 Code Definitions

**EFI_FONT_DISPLAY_INFO**

**Summary**

Describes font output-related information.

**Prototype**

```c
typedef struct _EFI_FONT_DISPLAY_INFO {
    EFI_GRAPHICS_OUTPUT_BLT_PIXEL ForegroundColor;
    EFI_GRAPHICS_OUTPUT_BLT_PIXEL BackgroundColor;
    EFI_FONT_INFO_MASK FontInfoMask;
    EFI_FONT_INFO FontInfo
} EFI_FONT_DISPLAY_INFO;
```
Members

FontInfo

The font information. Type `EFI_FONT_INFO` is defined in `EFI_HII_STRING_PROTOCOL.NewString()`.

ForegroundColor

The color of the “on” pixels in the glyph in the bitmap.

BackgroundColor

The color of the “off” pixels in the glyph in the bitmap.

FontInfoMask

The font information mask determines which portion of the font information will be used and what to do if the specific font is not available.

Description

This structure is used for describing the way in which a string should be rendered in a particular font. `FontInfo` specifies the basic font information and `ForegroundColor` and `BackgroundColor` specify the color in which they should be displayed. The flags in `FontInfoMask` describe where the system default should be supplied instead of the specified information. The flags also describe what options can be used to make a match between the font requested and the font available.

If `EFI_FONT_INFO_SYS_FONT` is specified, then the font name in `FontInfo` is ignored and the system font name is used. This flag cannot be used with `EFI_FONT_INFO_ANY_FONT`.

If `EFI_FONT_INFO_SYS_SIZE` is specified, then the font height specified in `FontInfo` is ignored and the system font height is used instead. This flag cannot be used with `EFI_FONT_INFO_ANY_SIZE`.

If `EFI_FONT_INFO_SYS_STYLE` is specified, then the font style in `FontInfo` is ignored and the system font style is used. This flag cannot be used with `EFI_FONT_INFO_ANY_STYLE`.

If `EFI_FONT_INFO_SYS_FORE_COLOR` is specified, then `ForegroundColor` is ignored and the system foreground color is used.

If `EFI_FONT_INFO_SYS_BACK_COLOR` is specified, then `BackgroundColor` is ignored and the system background color is used.

If `EFI_FONT_INFO_RESIZE` is specified, then the system may attempt to stretch or shrink a font to meet the size requested. This flag cannot be used with `EFI_FONT_INFO_ANY_SIZE`.

If `EFI_FONT_INFOorestyle` is specified, then the system may attempt to remove some of the specified styles in order to meet the style requested. This flag cannot be used with `EFI_FONT_INFO_ANY_STYLE`.

If `EFI_FONT_INFO_ANY_FONT` is specified, then the system may attempt to match with any font. This flag cannot be used with `EFI_FONT_INFO_SYS_FONT`.

If `EFI_FONT_INFO_ANY_SIZE` is specified, then the system may attempt to match with any font size. This flag cannot be used with `EFI_FONT_INFO_SYS_SIZE` or `EFI_FONT_INFO_RESIZE`. 
If **EFI_FONT_INFO_ANY_STYLE** is specified, then the system may attempt to match with any font style. This flag cannot be used with **EFI_FONT_INFO_SYS_STYLE** or **EFI_FONT_INFO_RESTYLE**.

**Related Definitions**

```c
typedef UINT32 EFI_FONT_INFO_MASK;
#define EFI_FONT_INFO_SYS_FONT    0x00000001
#define EFI_FONT_INFO_SYS_SIZE    0x00000002
#define EFI_FONT_INFO_SYS_STYLE   0x00000004
#define EFI_FONT_INFO_SYS_FORE_COLOR 0x00000010
#define EFI_FONT_INFO_SYS_BACK_COLOR 0x00000020
#define EFI_FONT_INFO_RESIZE      0x00001000
#define EFI_FONT_INFO_RESTYLE     0x00002000
#define EFI_FONT_INFO_ANY_FONT    0x00010000
#define EFI_FONT_INFO_ANY_SIZE    0x00020000
#define EFI_FONT_INFO_ANY_STYLE   0x00040000
```

**EFI_IMAGE_OUTPUT**

**Summary**

Describes information about either a bitmap or a graphical output device.

**Prototype**

```c
typedef struct _EFI_IMAGE_OUTPUT {
    UINT16 Width;
    UINT16 Height;
    union {
        EFI_GRAPHICS_OUTPUT_BLT_PIXEL *Bitmap;
        EFI_GRAPHICS_OUTPUT_PROTOCOL *Screen;
    }
    Image;
} EFI_IMAGE_OUTPUT;
```

**Members**

- **Width**
  Width of the output image.

- **Height**
  Height of the output image.

- **Bitmap**
  Points to the output bitmap.

- **Screen**
  Points to the **EFI_GRAPHICS_OUTPUT_PROTOCOL** which describes the screen on which to draw the specified string.
34.3 String Protocol

EFI_HII_STRING_PROTOCOL

Summary

Interfaces which manipulate string data.

GUID

```
#define EFI_HII_STRING_PROTOCOL_GUID   \
  { 0xfd96974, 0x23aa, 0x4cdc, \
    { 0xb9, 0xcb, 0x98, 0xd1, 0x77, 0x50, 0x32, 0x2a } }
```

Protocol

```
typedef struct _EFI_HII_STRING_PROTOCOL {
EFI_HII_NEW_STRING NewString;
EFI_HII_GET_STRING GetString;
EFI_HII_SET_STRING SetString;
EFI_HII_GET_LANGUAGES GetLanguages;
EFI_HII_GET_2ND_LANGUAGES GetSecondaryLanguages;
} EFI_HII_STRING_PROTOCOL;
```

Members

- **NewString**
  Add a new string.

- **GetString**
  Retrieve a string and related string information.

- **SetString**
  Change a string.

- **GetLanguages**
  List the languages for a particular package list.

- **GetSecondaryLanguages**
  List supported secondary languages for a particular primary language.

EFI_HII_STRING_PROTOCOL.NewString()

Summary

Creates a new string in a specific language and add it to strings from a specific package list.
Prototype

```
typedef
  EFI_STATUS
(EFIAPI *EFI_HII_NEW_STRING) (  
  IN CONST EFI_HII_STRING_PROTOCOL *This,
  IN EFI_HII_HANDLE PackageList,
  OUT EFI_STRING_ID *StringId
  IN CONST CHAR8 *Language,
  IN CONST CHAR16 *LanguageName OPTIONAL,
  IN CONST EFI_STRING String,
  IN CONST EFI_FONT_INFO *StringFontInfo
);
```

Parameters

**This**
A pointer to the **EFI_HII_STRING_PROTOCOL** instance.

**PackageList**
Handle of the package list where this string will be added.

**Language**
Points to the language for the new string. The language information is in the format described by Appendix M of the UEFI Specification.

**LanguageName**
Points to the printable language name to associate with the passed in **Language** field. This is analogous to passing in "zh-Hans" in the **Language** field and **LanguageName** might contain "Simplified Chinese" as the printable language.

**String**
Points to the new null-terminated string.

**StringFontInfo**
Points to the new string’s font information or NULL if the string should have the default system font, size and style.

**StringId**
On return, contains the new strings id, which is unique within **PackageList**. Type **EFI_STRING_ID** is defined in Section 33.3.8.2.1.

Description

This function adds the string **String** to the group of strings owned by **PackageList**, with the specified font information **StringFontInfo** and returns a new string id. The new string identifier is guaranteed to be unique within the package list. That new string identifier is reserved for all languages in the package list.
Related Definitions

typedef struct {
    EFI_HII_FONT_STYLE FontStyle;
    UINT16 FontSize;
    CHAR16 FontName[...];
} EFI_FONT_INFO;

FontStyle
The design style of the font. Type EFI_HII_FONT_STYLE is defined in Section 33.3.3.1.

FontSize
The character cell height, in pixels.

FontName
The null-terminated font family name.

Status Codes Returns

 EFI_SUCCESS The new string was added successfully
 EFI_OUT_OF_RESOURCES Could not add the string.
 EFI_INVALID_PARAMETER String is NULL or StringId is NULL or Language is NULL.
 EFI_NOT_FOUND The input package list could not be found in the current database.

EFI_HII_STRING_PROTOCOL.GetString()

Summary
Returns information about a string in a specific language, associated with a package list.

Prototype

typedef EFI_STATUS
    (EFIAPI *EFI_HII_STRING_PROTOCOL) (        
    IN   CONST EFI_HII_STRING_PROTOCOL  *This,
    IN   CONST CHAR8                   *Language,
    IN   EFI_HII_HANDLE                PackageList,
    IN   EFI_STRING_ID                 StringId,
    OUT  EFI_STRING                    String,
    IN OUT UINTN                       *StringLength,
    OUT  EFI_FONT_INFO                 **StringFontInfo OPTIONAL
    );

Parameters

    This
A pointer to the EFI_HII_STRING_PROTOCOL instance.
PackageList

The package list in the HII database to search for the specified string.

Language

Points to the language for the retrieved string. Callers of interfaces that require RFC 4646 language codes to retrieve a Unicode string must use the RFC 4647 algorithm to lookup the Unicode string with the closest matching RFC 4646 language code.

StringId

The string’s id, which is unique within PackageList.

String

Points to the new null-terminated string.

StringSize

On entry, points to the size of the buffer pointed to by String, in bytes. On return, points to the length of the string, in bytes.

StringFontInfo

Points to a buffer that will be callee allocated and will have the string's font information into this buffer. The caller is responsible for freeing this buffer. If the parameter is NULL a buffer will not be allocated and the string font information will not be returned.

Description

This function retrieves the string specified by StringId which is associated with the specified PackageList in the language Language and copies it into the buffer specified by String.

If the string specified by StringId is not present in the specified PackageList, then EFI_NOT_FOUND is returned. If the string specified by StringId is present, but not in the specified language then EFI_INVALID_LANGUAGE is returned.

If the buffer specified by StringSize is too small to hold the string, then EFI_BUFFER_TOO_SMALL will be returned. StringSize will be updated to the size of buffer actually required to hold the string.
Status Codes Returned

- **EFI_SUCCESS**: The string was returned successfully.
- **EFI_NOT_FOUND**: The string specified by `StringId` is not available. The specified `PackageList` is not in the Database.
- **EFI_INVALID_LANGUAGE**: The string specified by `StringId` is available but not in the specified language.
- **EFI_BUFFER_TOO_SMALL**: The buffer specified by `StringLength` is too small to hold the string.
- **EFI_INVALID_PARAMETER**: The `Language` or `StringLength` was NULL.
- **EFI_INVALID_PARAMETER**: The value referenced by `StringLength` was not zero and `String` was NULL.
- **EFI_OUT_OF_RESOURCES**: There were insufficient resources to complete the request.

EFI_HII_STRING_PROTOCOL::SetString()

**Summary**
Change information about the string.

**Prototype**
```c
typedef
EFI_STATUS
(EFIAPI *EFI_HII_SET_STRING) (  
    IN CONST EFI_HII_STRING_PROTOCOL *This,  
    IN EFI_HII_HANDLE PackageList,  
    IN EFI_STRING_ID StringId,  
    IN CONST CHAR8 *Language,  
    IN CONST EFI_STRING String,  
    IN CONST EFI_FONT_INFO *StringFontInfo OPTIONAL  
);
```

**Parameters**
- **This**
  A pointer to the `EFI_HII_STRING_PROTOCOL` instance.
- **PackageList**
  The package list containing the strings.
- **Language**
  Points to the language for the updated string.
- **StringId**
  The string id, which is unique within `PackageList`.
- **String**
  Points to the new null-terminated string.
**StringFontInfo**
Points to the string’s font information or NULL if the string font information is not changed.

**Description**
This function updates the string specified by `StringId` in the specified `PackageList` to the text specified by `String` and, optionally, the font information specified by `StringFontInfo`. There is no way to change the font information without changing the string text.

**Status Codes Returned**
- **EFI_SUCCESS** The string was successfully updated.
- **EFI_NOT_FOUND** The string specified by `StringId` is not in the database. The specified `PackageList` is not in the Database.
- **EFI_INVALID_PARAMETER** The `String` or `Language` was NULL.
- **EFI_OUT_OF_RESOURCES** The system is out of resources to accomplish the task.

**EFI_HII_STRING_PROTOCOL.GetLanguages()**

**Summary**
Returns a list of the languages present in strings in a package list.

**Prototype**
```c
typedef
EFI_STATUS
(EIFIAPI *EFI_HII_GET_LANGUAGES) (
    IN   CONST EFI_HII_STRING_PROTOCOL  *This,
    IN   EFI_HII_HANDLE                PackageList,
    IN OUT CHAR8                        *Languages,
    IN OUT UINTN                         *LanguagesSize
);```

**Parameters**
- **This**
  A pointer to the `EFI_HII_STRING_PROTOCOL` instance.
- **PackageList**
  The package list to examine.
- **Languages**
  Points to the buffer to hold the returned null-terminated ASCII string.
**LanguageSize**

On entry, points to the size of the buffer pointed to by **Languages**, in bytes. On return, points to the length of **Languages**, in bytes.

**Description**

This function returns the list of supported languages, in the format specified in Appendix M.

**Status Codes Returned**

- **EFI_SUCCESS**: The languages were returned successfully.
- **EFI_BUFFER_TOO_SMALL**: The **LanguageSize** is too small to hold the list of supported languages. **LanguageSize** is updated to contain the required size.
- **EFI_NOT_FOUND**: The specified **PackageList** is not in the Database.
- **EFI_INVALID_PARAMETER**: The value referenced by **LanguageSize** is null.

**EFI_HII_STRING_PROTOCOL.GetSecondaryLanguages()**

**Summary**

Given a primary language, returns the secondary languages supported in a package list.

**Prototype**

```c
typedef EFI_STATUS (EFIAPIC *EFI_HII_GET_2ND_LANGUAGES) (  
    IN   CONST EFI_HII_STRING_PROTOCOL  *This,  
    IN   EFI_HII_HANDLE   PackageList,  
    IN   CONST CHAR8*    PrimaryLanguage;  
    IN OUT CHAR8*   *SecondaryLanguages,  
    IN OUT UINTN    *SecondaryLanguagesSize
);  
```

**Parameters**

- **This**: A pointer to the **EFI_HII_STRING_PROTOCOL** instance.
- **PackageList**: The package list to examine.
- **PrimaryLanguage**: Points to the null-terminated ASCII string that specifies the primary language.

Languages are specified in the format specified in Appendix M of the UEFI Specification.
SecondaryLanguages
Points to the buffer to hold the returned null-terminated ASCII string that describes the list of secondary languages for the specified PrimaryLanguage. If there are no secondary languages, the function returns successfully, but this is set to NULL.

SecondaryLanguagesSize
On entry, points to the size of the buffer pointed to by SecondaryLanguages, in bytes. On return, points to the length of SecondaryLanguages in bytes.

Description
Each string package has associated with it a single primary language and zero or more secondary languages. This routine returns the secondary languages associated with a package list.

Status Codes Returned
- EFI_SUCCESS: Secondary languages correctly returned
- EFI_BUFFER_TOO_SMALL: The buffer specified by SecondaryLanguagesSize is too small to hold the returned information. SecondaryLanguagesSize is updated to hold the size of the buffer required.
- EFI_INVALID_LANGUAGE: The language specified by FirstLanguage is not present in the specified package list.
- EFI_NOT_FOUND: The specified PackageList is not in the Database.
- EFI_INVALID_PARAMETER: PrimaryLanguage or SecondaryLanguagesSize is NULL.
- EFI_INVALID_PARAMETER: The value referenced by SecondaryLanguagesSize is not zero and SecondaryLanguages is NULL.

34.4 Image Protocol

EFI_HII_IMAGE_PROTOCOL

Summary
Protocol which allow access to images in the images database.

GUID
#define EFI_HII_IMAGE_PROTOCOL_GUID\
{ 0x31a6406a, 0x6bdf, 0x4e46,\
  { 0xb2, 0xa2, 0xeb, 0xaa, 0x89, 0xc4, 0x9, 0x20 } }
Protocol

typedef struct _EFI_HII_IMAGE_PROTOCOL {
    EFI_HII_NEW_IMAGE NewImage;
    EFI_HII_GET_IMAGE GetImage;
    EFI_HII_SET_IMAGE SetImage;
    EFI_HII_DRAW_IMAGE DrawImage;
    EFI_HII_DRAW_IMAGE_ID DrawImageId;
} EFI_HII_IMAGE_PROTOCOL;

Members

    NewImage
    Add a new image.

    GetImage
    Retrieve an image and related font information.

    SetImage
    Change an image.

EFI_HII_IMAGE_PROTOCOL.NewImage()

Summary
Creates a new image and add it to images from a specific package list.

Prototype

typedef
    EFI_STATUS
    (EFIAPI *EFI_HII_NEW_IMAGE) (
        IN CONST EFI_HII_IMAGE_PROTOCOL *This,
        IN EFI_HII_HANDLE PackageList,
        OUT EFI_IMAGE_ID *ImageId
        IN CONST EFI_IMAGE_INPUT *Image
    );

Parameters

    This
    A pointer to the EFI_HII_IMAGE_PROTOCOL instance.

    PackageList
    Handle of the package list where this image will be added.

    ImageId
    On return, contains the new image id, which is unique within PackageList.

    Image
    Points to the image.
Description
This function adds the image `Image` to the group of images owned by `PackageList`, and returns a new image identifier (`ImageId`).

Related Definitions
```c
typedef UINT16 EFI_IMAGE_ID;
typedef struct 
    {
        UINT32 Flags;
        UINT16 Width;
        UINT16 Height;
        EFI_GRAPHICS_OUTPUT_BLT_PIXEL *Bitmap;
    } EFI_IMAGE_INPUT;
```

Flags
Describe image characteristics. If `EFI_IMAGE_TRANSPARENT` is set, then the image was designed for transparent display.

```c
#define EFI_IMAGE_TRANSPARENT  0x00000001
```

Width
Image width, in pixels.

Height
Image height, in pixels.

Bitmap
A pointer to the actual bitmap, organized left-to-right, top-to-bottom. The size of the bitmap is `Width * Height * sizeof(EFI_GRAPHICS_OUTPUT_BLT_PIXEL)`.

Status Codes Returns
- `EFI_SUCCESS` The new image was added successfully
- `EFI_OUT_OF_RESOURCES` Could not add the image.
- `EFI_INVALID_PARAMETER` `Image` is NULL or `ImageId` is NULL.
- `EFI_NOT_FOUND` The `PackageList` could not be found.

`EFI_HII_IMAGE_PROTOCOL.GetImage()`

Summary
Returns information about an image, associated with a package list.
Prototype

```c
typedef EFI_STATUS (EFIAPI *EFI_HII_GET_IMAGE) (
    IN CONST EFI_HII_IMAGE_PROTOCOL *This,
    IN EFI_HII_HANDLE PackageList,
    IN EFI_IMAGE_ID ImageId,
    OUT EFI_IMAGE_INPUT *Image
);
```

Parameters

- `This`  
  A pointer to the `EFI_HII_IMAGE_PROTOCOL` instance.
- `PackageList`  
  The package list in the HII database to search for the specified image.
- `ImageId`  
  The image’s id, which is unique within `PackageList`.
- `Image`  
  Points to the new image.

Description

This function retrieves the image specified by `ImageId` which is associated with the specified `PackageList` and copies it into the buffer specified by `Image`.

If the image specified by `ImageId` is not present in the specified `PackageList`, then `EFI_NOT_FOUND` is returned.

The actual bitmap (`Image->Bitmap`) should not be freed by the caller and should not be modified directly.

Status Codes Returned

- `EFI_SUCCESS`  
  The image was returned successfully.
- `EFI_NOT_FOUND`  
  The image specified by `ImageId` is not available. The specified `PackageList` is not in the Database.
- `EFI_INVALID_PARAMETER`  
  `Image` was NULL.
- `EFI_OUT_OF_RESOURCES`  
  The bitmap could not be retrieved because there was not enough memory.

`EFI_HII_IMAGE_PROTOCOL.SetImage()`

Summary

Change information about the image.
Prototype

typedef
EFI_STATUS
(EIFIAPI *EFI_HII_SET_IMAGE) (
    IN CONST EFI_HII_IMAGE_PROTOCOL *This,
    IN EFI_HII_HANDLE PackageList,
    IN EFI_IMAGE_ID ImageId,
    IN CONST EFI_IMAGE_INPUT *Image,
);

Parameters

This
    A pointer to the EFI_HII_IMAGE_PROTOCOL instance.

PackageList
    The package list containing the images.

ImageId
    The image id, which is unique within PackageList.

Image
    Points to the image.

Description
This function updates the image specified by ImageId in the specified PackageListHandle to the image specified by Image.

Status Codes Returned

EFI_SUCCESS The image was successfully updated.
EFI_NOT_FOUND The image specified by ImageId is not in the database. The specified PackageList is not in the Database.
EFI_INVALID_PARAMETER The Image was NULL.

EFI_HII_IMAGE_PROTOCOL.DrawImage()

Summary
Renders an image to a bitmap or to the display.
Prototype

typedef
  EFI_STATUS
  (EFI_APIC *EFI_HII_DRAW_IMAGE) (  
    IN CONST EFI_HII_IMAGE_PROTOCOL *This,
    IN EFI_HII_DRAW_FLAGS Flags,
    IN CONST EFI_IMAGE_INPUT *Image,
    IN OUT EFI_IMAGE_OUTPUT **Blt,
    IN UINTN BltX,
    IN UINTN BltY,
  );

Parameters

  This

    A pointer to the EFI_HII_IMAGE_PROTOCOL instance.

  Flags

    Describes how the image is to be drawn. EFI_HII_DRAW_FLAGS is defined in Related Definitions, below.

  Image

    Points to the image to be displayed.

  Blt

    If this points to a non-NULL on entry, this points to the image, which is Width pixels wide and Height pixels high. The image will be drawn onto this image and EFI_HII_DRAW_FLAG_CLIP is implied. If this points to a NULL on entry, then a buffer will be allocated to hold the generated image and the pointer updated on exit. It is the caller’s responsibility to free this buffer.

    BltX, BltY

    Specifies the offset from the left and top edge of the image of the first pixel in the image.

Description

This function renders an image to a bitmap or the screen using the specified color and options. It draws the image on an existing bitmap, allocates a new bitmap or uses the screen. The images can be clipped.

If EFI_HII_DRAW_FLAG_CLIP is set, then all pixels drawn outside the bounding box specified by Width and Height are ignored.

The EFI_HII_DRAW_FLAG_TRANSPARENT flag determines whether the image will be drawn transparent or opaque. If EFI_HII_DRAW_FLAG_FORCE_TRANS is set then the image’s pixels will be drawn so that all “off” pixels in the image will be drawn using the pixel value from BLT and all other pixels will be copied. If EFI_HII_DRAW_FLAG_FORCE_OPAQUE is set, then the image’s pixels will be copied directly to the destination. If EFI_HII_DRAW_FLAG_DEFAULT is set, then the image will be drawn
transparently or opaque, depending on the image’s transparency setting (see `EFI_IMAGE_TRANSPARENT`). Images cannot be drawn transparently if `Blt` is NULL.

If `EFI_HII_DIRECT_TO_SCREEN` is set, then the image will be written directly to the output device specified by `Screen`. Otherwise the image will be rendered to the bitmap specified by `Bitmap`.

**Status Codes Returned**

- **EFI_SUCCESS** The image was successfully updated.
- **EFI_OUT_OF_RESOURCES** Unable to allocate an output buffer for `Blt`.
- **EFI_INVALID_PARAMETER** The `Image` or `Blt` was NULL.

**EFI_HII_IMAGE_PROTOCOL.DrawLineId()**

**Summary**

Render an image to a bitmap or the screen containing the contents of the specified image.

**Prototype**

```c
typedef EFI_STATUS (EFIAPI *EFI_HII_DRAW_IMAGE_ID) (
    IN CONST EFI_HII_IMAGE_PROTOCOL *This,
    IN EFI_HII_DRAW_FLAGS Flags,
    IN EFI_HII_HANDLE PackageList,
    IN EFI_IMAGE_ID ImageId,
    IN OUT EFI_IMAGE_OUTPUT **Blt,
    IN UINTN BltX,
    IN UINTN BltY,
);
```

**Parameters**

- **This**
  
  A pointer to the `EFI_HII_IMAGE_PROTOCOL` instance.

- **Flags**
  
  Describes how the image is to be drawn. `EFI_HII_DRAW_FLAGS` is defined in Related Definitions, below.

- **PackageList**
  
  The package list in the HII database to search for the specified image.

- **ImageId**
  
  The image’s id, which is unique within `PackageList`.

- **Blt**
  
  If this points to a non-NULL on entry, this points to the image, which is `Width` pixels wide and `Height` pixels high. The image will be drawn onto this image and
**EFI_HII_DRAW_FLAG_CLIP** is implied. If this points to a NULL on entry, then a buffer will be allocated to hold the generated image and the pointer updated on exit. It is the caller’s responsibility to free this buffer.

*Bltx, Blty*

Specifies the offset from the left and top edge of the output image of the first pixel in the image.

**Description**

This function renders an image to a bitmap or the screen using the specified color and options. It draws the image on an existing bitmap, allocates a new bitmap or uses the screen. The images can be clipped.

If **EFI_HII_DRAW_FLAG_CLIP** is set, then all pixels drawn outside the bounding box specified by **Width** and **Height** are ignored.

The **EFI_HII_DRAW_FLAG_TRANSPARENT** flag determines whether the image will be drawn transparent or opaque. If **EFI_HII_DRAW_FLAG_FORCE_TRANS** is set, then the image will be drawn so that all “off” pixels in the image will be drawn using the pixel value from **Blt** and all other pixels will be copied. If **EFI_HII_DRAW_FLAG_FORCE_OPAQUE** is set, then the image’s pixels will be copied directly to the destination. If **EFI_HII_DRAW_FLAG_DEFAULT** is set, then the image will be drawn transparently or opaque, depending on the image’s transparency setting (see **EFI_IMAGE_TRANSPARENT**). Images cannot be drawn transparently if **Blt** is NULL.

If **EFI_HII_DIRECT_TO_SCREEN** is set, then the image will be written directly to the output device specified by **Screen**. Otherwise the image will be rendered to the bitmap specified by **Bitmap**.

**Related Definitions**

```c
typedef UINT32 EFI_HII_DRAW_FLAGS;
#define EFI_HII_DRAW_FLAG_CLIP           0x00000001
#define EFI_HII_DRAW_FLAG_TRANSPARENT    0x00000030
#define EFI_HII_DRAW_FLAG_DEFAULT       0x00000000
#define EFI_HII_DRAW_FLAG_FORCE_TRANS   0x00000010
#define EFI_HII_DRAW_FLAG_FORCE_OPAQUE  0x00000020
#define EFI_HII_DIRECT_TO_SCREEN         0x00000080
```

**Status Codes Returned**

- **EFI_SUCCESS** The image was successfully updated.
- **EFI_OUT_OF_RESOURCES** Unable to allocate an output buffer for **RowInfoArray** or **Blt**.
- **EFI_NOT_FOUND** The image specified by **ImageId** is not in the database. The specified **PackageList** is not in the Database.
- **EFI_INVALID_PARAMETER** The **Image** or **Blt** was NULL.

### 34.5 EFI HII Image Ex Protocol

The EFI HII Image Ex protocol defines an extension to the EFI HII Image protocol which enables various new capabilities described in this section.
EFI_HII_IMAGE_EX_PROTOCOL

Summary
Protocol which allows access to the images in the images database

GUID
#define EFI_HII_IMAGE_EX_PROTOCOL_GUID \
{0x1a1241e6, 0x8f19, 0x41a9, 0xbc, \
{0xe, 0xe8, 0xef,0x39, 0xe0, 0x65, 0x46}

Protocol
typedef struct _EFI_HII_IMAGE_EX_PROTOCOL {
  EFI_HII_NEW_IMAGE_EX    NewImageEx;
  EFI_HII_GET_IMAGE_EX    GetImageEx;
  EFI_HII_SET_IMAGE_EX    SetImageEx;
  EFI_HII_DRAW_IMAGE_EX   DrawImageEx;
  EFI_HII_DRAW_IMAGE_ID_EX  DrawImageIdEx;
  EFI_HII_GET_IMAGE_INFO  GetImageInfo;
} EFI_HII_IMAGE_EX_PROTOCOL;

Members
  NewImageEx          Add a new image. This protocol invokes the original 
                     EFI_HII_IMAGE_PROTOCOL.NewImage() implicitly.
  GetImageEx          Retrieve an image and the related image information. This 
                     function will try to locate the 
                     EFI_HII_IMAGE_DECODER_PROTOCOL if the image decoder for 
                     the image is not supported by the EFI HII image EX protocol.
  SetImageEx          Change information about the image, this protocol invokes original 
                     EFI_HII_IMAGE_PROTOCOL.SetImage() implicitly.
  DrawImageEx         Renders an image to a bitmap or the display, this protocol 
                     invokes original EFI_HII_IMAGE_PROTOCOL.DrawImage() implicitly.
  DrawImageIdEx       Renders an image to a bitmap or the screen containing the 
                     contents of the specified image, this protocol invokes original 
                     EFI_HII_IMAGE_PROTOCOL.DrawImageId() implicitly.
  GetImageInfo        This function retrieves the image information specified by the 
                     image ID which is associated with the specified HII package 
                     list. This function only returns the geometry of the image 
                     instead of allocating the memory buffer and decoding the 
                     image to the buffer.

EFI_HII_IMAGE_EX_PROTOCOL.NewImageEx()

Summary
The prototype of this extension function is the same with EFI_HII_IMAGE_PROTOCOL.NewImage().
Protocol

typedef
EFI_STATUS
(EIFIAPI *EFI_HII_NEW_IMAGE_EX)(
    IN CONST EFI_HII_IMAGE_EX_PROTOCOL *This,
    IN EFI_HII_HANDLE PackageList,
    OUT EFI_IMAGE_ID *ImageId
    IN OUT EFI_IMAGE_INPUT *Image
    );

Parameters
Same with EFI_HII_IMAGE_PROTOCOL.NewImage().

Description
Same with EFI_HII_IMAGE_PROTOCOL.NewImage(). This protocol invokes EFI_HII_IMAGE_PROTOCOL.NewImage() implicitly.

Status Codes Returned
Same as EFI_HII_IMAGE_PROTOCOL.NewImage().

EFI_HII_IMAGE_EX_PROTOCOL.GetImageEx()

Summary
Return the information about the image, associated with the package list. The prototype of this extension function is the same with EFI_HII_IMAGE_PROTOCOL.GetImage().

Protocol

typedef
EFI_STATUS
(EIFIAPI *EFI_HII_GET_IMAGE_EX)(
    IN CONST EFI_HII_IMAGE_EX_PROTOCOL *This,
    IN EFI_HII_HANDLE PackageList,
    IN EFI_IMAGE_ID ImageId,
    OUT EFI_IMAGE_INPUT *Image
    );

Parameters
Same with EFI_HII_IMAGE_PROTOCOL.GetImage()

Description
This function is similar to EFI_HII_IMAGE_PROTOCOL.GetImage(). The difference is that this function will locate all EFI_HII_IMAGE_DECODER_PROTOCOL instances installed in the system if the decoder of the certain image type is not supported by the EFI_HII_IMAGE_EX_PROTOCOL. The function will
attempt to decode the image to the `EFI_IMAGE_INPUT` using the first `EFI_HII_IMAGE_DECODER_PROTOCOL` instance that supports the requested image type.

**Status Codes Returned**

Same as `EFI_HII_IMAGE_PROTOCOL.GetImage()`.

### EFI_HII_IMAGE_EX_PROTOCOL.SetImageEx()

**Summary**

Change the information about the image. The prototype of this extension function is the same with `EFI_HII_IMAGE_PROTOCOL.SetImage()`.

**Protocol**

```c
typedef EFI_STATUS
  (EFIAPI *EFI_HII_SET_IMAGE_EX)(
    IN CONST EFI_HII_IMAGE_EX_PROTOCOL *This,
    IN EFI_HII_HANDLE PackageList,
    IN EFI_IMAGE_ID ImageId,
    IN CONST EFI_IMAGE_INPUT *Image
  );
```

**Parameters**

Same with `EFI_HII_IMAGE_PROTOCOL.SetImage()`.

**Description**

Same with `EFI_HII_IMAGE_PROTOCOL.SetImage()`, this protocol invokes `EFI_HII_IMAGE_PROTOCOL.SetImage()` implicitly.

**Status Codes Returned**

Same as `EFI_HII_IMAGE_PROTOCOL.SetImage()`.

### EFI_HII_IMAGE_EX_PROTOCOL.DrawImageEx()

**Summary**

Renders an image to a bitmap or to the display. The prototype of this extension function is the same with `EFI_HII_IMAGE_PROTOCOL.DrawImage()`.
Protocol

typedef

EFI_STATUS

(EFIAPI *EFI_HII_DRAW_IMAGE_EX)(

    IN CONST EFI_HII_IMAGE_EX_PROTOCOL *This,
    IN EFI_HII_DRAW_FLAGS Flags,
    IN CONST EFI_IMAGE_INPUT *Image,
    IN OUT EFI_IMAGE_OUTPUT **Blt,
    IN UINTN BltX,
    IN UINTN BltY
)

Parameters

Same with EFI_HII_IMAGE_PROTOCOL.DrawImageId().

Description

Same with EFI_HII_IMAGE_PROTOCOL.DrawImage(), this protocol invokes

    EFI_HII_IMAGE_PROTOCOL.DrawImage() implicitly.

Status Codes Returned

Same as EFI_HII_IMAGE_PROTOCOL.DrawImage().

EFI_HII_IMAGE_EX_PROTOCOL.DrawImageIdEx() 

Summary

Renders an image to a bitmap or the screen containing the contents of the specified image. The prototype of this extension function is the same with EFI_HII_IMAGE_PROTOCOL.DrawImageId().
Description

This function is similar to EFI_HII_IMAGE_PROTOCOL.DrawImageId(). The difference is that this function will locate all EFI_HII_IMAGE_DECODER_PROTOCOL instances installed in the system if the decoder of the certain image type is not supported by the EFI_HII_IMAGE_EX_PROTOCOL. The function will attempt to decode the image to the EFI_IMAGE_INPUT using the first EFI_HII_IMAGE_DECODER_PROTOCOL instance that supports the requested image type.

Status Codes Returned

Same as EFI_HII_IMAGE_PROTOCOL.DrawImageId().

EFI_HII_IMAGE_EX_PROTOCOL.GetImageInfo()

Summary

The function returns the information of the image. This function is differ from the EFI_HII_IMAGE_EX_PROTOCOL.GetImageEx() This function only returns the geometry of the image instead of decoding the image to the buffer.

Protocol

```c
typedef EFI_STATUS
(EFIAPI *EFI_HII_GET_IMAGE_INFO)(
    IN CONST EFI_HII_IMAGE_EX_PROTOCOL *This,
    IN EFI_HII_HANDLE PackageList,
    IN EFI_IMAGE_ID ImageId,
    OUT EFI_IMAGE_OUTPUT *Image
);
```

Parameters

- **This**: EFI_HII_IMAGE_EX_PROTOCOL instance.
- **PackageList**: The HII package list.
- **ImageId**: The HII image ID.
- **Image**: EFI_IMAGE_OUTPUT to retrieve the image information. Only `Image.Width` and `Image.Height` will be updated by this function. `Image.Bitmap` is always set to NULL.

Description

This function returns the image information to EFI_IMAGE_OUTPUT. Only the width and height are returned to the EFI_IMAGE_OUTPUT instead of decoding the image to the buffer. This function is used to get the geometry of the image. This function will try to locate all of the EFI_HII_IMAGE_DECODER_PROTOCOL installed on the system if the decoder of image type is not supported by the EFI_HII_IMAGE_EX_PROTOCOL.
### Status Codes Returned

<table>
<thead>
<tr>
<th>Code</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>EFI_SUCCESS</td>
<td>The image information was returned to Image.</td>
</tr>
<tr>
<td>EFI_OUT_OF_RESOURCES</td>
<td>Memory allocation failed in this function.</td>
</tr>
<tr>
<td>EFI_UNSUPPORTED</td>
<td>The format of image is not supported.</td>
</tr>
<tr>
<td>EFI_NOT_FOUND</td>
<td>The image was not found in the database.</td>
</tr>
<tr>
<td>EFI_INVALID_PARAMETER</td>
<td>The Image was NULL or ImageId was 0.</td>
</tr>
</tbody>
</table>

#### 34.6 EFI HII Image Decoder Protocol

For those HII image block types which don’t have the corresponding image decoder supported in EFI HII image EX protocol, `EFI_HII_IMAGE_DECODER_PROTOCOL` can be used to provide the proper image decoder. There may be more than one `EFI_HII_IMAGE_DECODER_PROTOCOL` instance installed in the system. Each image decoder can decode more than one HII image block types. Whether or not the HII image block type of image is supported by the certain image decoder is reported through the `EFI_HII_IMAGE_DECODER_PROTOCOL` `GetImageDecoderName()` method. Caller can invoke this function to verify the image is supported by the image decoder before sending the image raw data to the image decoder. There are two image decoder names defined in this specification: `EFI_HII_IMAGE_DECODER_NAME_JPEG` and `EFI_HII_IMAGE_DECODER_NAME_PNG`.

The image decoder protocol can publish the support for additional image decoder names other than the ones defined in this specification. This allows the image decoder to support additional image formats that are not defined by the HII image block types. In that case, callers can send the image raw data to the image decoder protocol instance to retrieve the image information or decode the image. Since the HII image block type of such images is not defined, the image may or may not be decoded by that decoder. The decoder can use the signature or data structures in the image raw data is check the format before it processes the image.

The `EFI_HII_IMAGE_EX_PROTOCOL` uses `EFI_HII_IMAGE_DECODER_PROTOCOL` as follows:
Figure 34-2 How EFI_HII_IMAGE_EX_PROTOCOL uses EFI_HII_IMAGE_DECODER_PROTOCOL

**EFI_HII_IMAGE_DECODER_PROTOCOL.DecodeImage()**

**Summary**
Provides the image decoder for specific image file formats.

**GUID**
```c
#define EFI_HII_IMAGE_DECODER_PROTOCOL_GUID 
{0x9E66F251, 0x727C, 0x418C, 
{0xBF, 0xD6, 0xC2, 0xB4, 0x25, 0x28, 0x18, 0xEA}}
```

**Protocol**
```c
typedef struct _EFI_HII_IMAGE_DECODER_PROTOCOL {
    EFI_HII_IMAGE_DECODER_GET_NAME GetImageDecoderName;
    EFI_HII_IMAGE_DECODER_GET_IMAGE_INFO GetImageInfo;
    EFI_HII_IMAGE_DECODER_DECODER GetName;
} EFI_HII_IMAGE_DECODER_PROTOCOL;
```

**Members**
GetImageDecodeName — The function returns the decoder name.
GetImageInfo — The function returns the image information
DecodeImage — The function decodes the image to the `EFI_IMAGE_INPUT`

### Status Codes Returned

<table>
<thead>
<tr>
<th>Status Code</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>EFI_SUCCESS</td>
<td>The image information was returned to <code>Bitmap</code>.</td>
</tr>
<tr>
<td>EFI_UNSUPPORTED</td>
<td>The image decoder can't decode this image.</td>
</tr>
<tr>
<td>EFI_OUT_OF_RESOURCE</td>
<td>Not enough memory to decode this image.</td>
</tr>
<tr>
<td>EFI_INVALID_PARAMETER</td>
<td>The <code>Image</code> was NULL or <code>ImageRawDataSize</code> was 0.</td>
</tr>
</tbody>
</table>

#### EFI_HII_IMAGE_DECODER_PROTOCOL.GetImageDecoderName()

**Summary**

This function returns the decoder name.

**Protocol**

```c
typedef EFI_STATUS (EFIAPICALLTYPE EFI_HII_IMAGE_DECODER_GET_NAME)(
    IN CONST EFI_HII_IMAGE_DECODER_PROTOCOL *This,
    IN OUT EFI_GUID **DecoderName,
    OUT UINT16 *NumberOfDecoderName
);
```

**Parameters**

- **This** — `EFI_HII_IMAGE_DECODER_PROTOCOL` instance.
- **DecoderName** — Pointer to a dimension to retrieve the decoder names in `EFI_GUID` format. The number of the decoder names is returned in `NumberOfDecoderName`.
- **NumberOfDecoderName** — Pointer to retrieve the number of decoders which supported by this decoder driver.

**Description**

There could be more than one `EFI_HII_IMAGE_DECODER_PROTOCOL` instances installed in the system for different image formats. This function returns the decoder name which callers can use to find the proper image decoder for the image. It is possible to support multiple image formats in one `EFI_HII_IMAGE_DECODER_PROTOCOL`. The capability of the supported image formats is returned in `DecoderName` and `NumberOfDecoderName`. 
Related Definitions

```c
//******************************************************************
// EFI_HII_IMAGE_DECODER_NAME
//******************************************************************
#define EFI_HII_IMAGE_DECODER_NAME_JPEG_GUID
{0xefefd093, 0xd9b, 0x46eb, 0xa8, 
  {0x56, 0x48, 0x35, 0x7, 0x0, 0xc9, 0x8}}

#define EFI_HII_IMAGE_DECODER_NAME_PNG_GUID
{0xaf060190, 0x5e3a, 0x4025, 0xaf, 
  {0xbd, 0xe1, 0xf9, 0x5, 0xbf, 0xaa, 0x4c}}
```

Status Codes Returned

<table>
<thead>
<tr>
<th>Code</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>EFI_SUCCESS</td>
<td>The image decoder names were returned in DecoderName.</td>
</tr>
<tr>
<td>EFI_UNSUPPORTED</td>
<td>No image decoders found in this EFI_HII_IMAGE_DECODER instance.</td>
</tr>
</tbody>
</table>

EFI_HII_IMAGE_DECODER_PROTOCOL.GetImageInfo()

Summary
The function returns the EFI_HII_IMAGE_DECODER_IMAGE_INFO to the caller.

Protocol

```c
typedef EFI_STATUS
(EIFIAPI *EFI_HII_IMAGE_DECODER_GET_IMAGE_INFO)(
  IN CONST EFI_HII_IMAGE_DECODER_PROTOCOL *This, 
  IN VOID *Image, 
  IN UINTN SizeOfImage, 
  IN OUT EFI_HII_IMAGE_DECODER_IMAGE_INFO **ImageInfo
);
```

Parameters

- **This**: EFI_HII_IMAGE_DECODER_PROTOCOL instance.
- **Image**: Pointer to the image raw data
- **SizeOfImage**: Size of the entire image raw data
- **ImageInfo**: Pointer to receive the EFI_HII_IMAGE_DECODER_IMAGE_INFO

Description
This function returns the image information of the given image raw data. This function first checks whether the image raw data is supported by this decoder or not. This function may go through the first few bytes in the image raw data for the specific data structure or the image signature. If the image is not supported by this image decoder, this function returns EFI_UNSUPPORTED to the caller. Otherwise, this
function returns the proper image information to the caller. It is the caller’s responsibility to free the `ImageInfo`.

**Status Codes Returned**

<table>
<thead>
<tr>
<th>Code</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>EFI_SUCCESS</td>
<td>The image information was returned to <code>ImageInfo</code>.</td>
</tr>
<tr>
<td>EFI_UNSUPPORTED</td>
<td>No image decoder for the given <code>Image</code> or the image decoder can't decode this image.</td>
</tr>
<tr>
<td>EFI_OUT_OF_RESOURCE</td>
<td>Not enough memory to decode this image for getting the image information.</td>
</tr>
<tr>
<td>EFI_INVALID_PARAMETER</td>
<td>The <code>Image</code> was NULL, <code>SizeOfImage</code> was 0 or the image is corrupted.</td>
</tr>
</tbody>
</table>

**Related Definitions**

```c
// ******************* EDFI_HII_IMAGE_DECODER_COLOR_TYPE *******************
typedef enum {
  EFI_HII_IMAGE_DECODER_COLOR_TYPE_RGB = 0,
  EFI_HII_IMAGE_DECODER_COLOR_TYPE_RGBA = 1,
  EFI_HII_IMAGE_DECODER_COLOR_TYPE_CMYK = 2,
  EFI_HII_IMAGE_DECODER_COLOR_TYPE_UNKNOWN = 0xff,
} EFI_HII_IMAGE_DECODER_COLOR_TYPE;
```

```c
// ******************* EFI_HII_IMAGE_DECODER_IMAGE_INFO_HEADER *******************
typedef struct _EFI_HII_IMAGE_DECODER_IMAGE_INFO_HEADER {
  EFI_GUID DecoderName;
  UINT16 ImageInfoSize;
  UINT16 ImageWidth;
  UINT16 ImageHeight;
  EFI_HII_IMAGE_DECODER_COLOR_TYPE ColorType;
  UINT8 ColorDepthInBits;
} EFI_HII_IMAGE_DECODER_IMAGE_INFO_HEADER;
```

DecoderName: Decoder Name
ImageInfoSize: The size of entire image information structure in bytes.
ImageWidth: The image width.
ImageHeight: The image height.
ColorType: The color type, refer to `EFI_HII_IMAGE_DECODER_COLOR_TYPE`.
ColorDepthInBits: The color depth in bits.
typedef struct _EFI_HII_IMAGE_DECODER_JPEG_INFO {
    EFI_HII_IMAGE_DECODER_IMAGE_INFO_HEADER Header;
    UINT16 ScanType;
    UINT64 Reserved;
} EFI_HII_IMAGE_DECODER_JPEG_INFO;

// EFI_HII_IMAGE_DECODER_JPEG_INFO

typedef struct _EFI_HII_IMAGE_DECODER_PNG_INFO {
    EFI_HII_IMAGE_DECODER_IMAGE_INFO_HEADER Header;
    UINT16 Channels;
    UINT64 Reserved;
} EFI_HII_IMAGE_DECODER_PNG_INFO;

// EFI_HII_IMAGE_DECODER_PNG_INFO

typedef struct _EFI_HII_IMAGE_DECODER_OTHER_INFO {
    EFI_HII_IMAGE_DECODER_IMAGE_INFO_HEADER Header;
    CHAR16 ImageExtension[1];
} EFI_HII_IMAGE_DECODER_OTHER_INFO;

// EFI_HII_IMAGE_DECODER_OTHER_INFO

Header EFI_HII_IMAGE_DECODER_IMAGE_INFO_HEADER
ScanType The scan type of the JPEG image
#define EFI_IMAGE_JPEG_SCANTYPE_PROGRESSIVE 0x01
#define EFI_IMAGE_JPEG_SCANTYPE_INTERNACE 0x02

Reserved Reserved

Channels Number of channels in the PNG image.

Reserved Reserved

ImageExtension The string of the image file extension. For example, "GIF", "TIFF" or others.
**EFI_HII_IMAGE_DECODER_PROTOCOL.Decode()**

**Summary**
The function decodes the image

**Protocol**
```c
typedef EFI_STATUS
(EFIAPI *EFI_HII_IMAGE_DECODER_DECODER)(
    IN CONST EFI_HII_IMAGE_DECODER_PROTOCOL *This,
    IN VOID *Image,
    IN UINTN ImageRawDataSize,
    IN OUT EFI_IMAGE_OUTPUT **Bitmap,
    IN BOOLEAN Transparent
);
```

**Parameters**
- **This**: EFI_HII_IMAGE_DECODER_PROTOCOL instance.
- **Image**: Pointer to the image raw data
- **ImageRawDataSize**: Size of the entire image raw data
- **Bitmap**: EFI_IMAGE_OUTPUT to receive the image or overlap the image on the original buffer.
- **Transparent**: BOOLEAN value indicates whether the image decoder has to handle the transparent image or not.

**Description**
This function decodes the image which the image type of this image is supported by this EFI_HII_IMAGE_DECODER_PROTOCOL. If *Bitmap* is not NULL, the caller intends to put the image in the given image buffer. That allows the caller to put an image overlap on the original image. The transparency is handled by the image decoder because the transparency capability depends on the image format. Callers can set **Transparent** to FALSE to force disabling the transparency process on the image. Forcing **Transparent** to FALSE may also improve the performance of the image decoding because the image decoder can skip the transparency processing.

If *Bitmap* is NULL, the image decoder allocates the memory buffer for the EFI_IMAGE_OUTPUT and decodes the image to the image buffer. It is the caller’s responsibility to free the memory for EFI_IMAGE_OUTPUT. Image decoder doesn’t have to handle the transparency in this case because there is no background image given by the caller. The background color in this case is all black (#00000000).

### 34.7 Font Glyph Generator Protocol

The EFI HII Font glyph generator protocol generates font glyphs of the requested characters according to the given font information. This protocol is utilized by the EFI_HII_FONT_EX_PROTOCOL when the character can’t be found in the existing glyph database. That is when glyph is not found in any HII font package,
EFI_HII_FONT_EX_PROTOCOL locates EFI_HII_FONT_GLYPH_GENERATOR_PROTOCOL to generate glyph block and insert glyph block into HII font package. The HII font package can be an existing HII font package or a new HII font package. This protocol can be provided by any driver that knows how to generate the glyph for a specific font family. For example, EFI application or driver may provide "Times new roman" font glyph generator driver. With this driver, platform can have "Times new roman" font supported on system.

**EFI_HII_FONT_GLYPH_GENERATOR_PROTOCOL**

**Summary**

EFI HII Font glyph generator protocol generates the glyph of the character according to the given font information.

**GUID**

```c
#define EFI_HII_FONT_GLYPH_GENERATOR_PROTOCOL_GUID 
  { 0xf7102853, 0x7787, 0x4dc2, 0xa8, 
    {0xa8, 0x21, 0xb5, 0xdd, 0x5, 0xc8, 0x9b } }
```

**Protocol**

```c
typedef struct _EFI_HII_FONT_GLYPH_GENERATOR_PROTOCOL {
  EFI GENERATE_GLYPH GenerateGlyph;
  EFI GENERATE_IMAGE GenerateGlyphImage;
} EFI_HII_FONT_GLYPH_GENERATOR_PROTOCOL;
```

**Members**

- **GenerateGlyph** The function generates the glyph information according to the given font information.
- **GenerateGlyphImage** The function generates the glyph image according to the given font information.

**EFI_HII_FONT_GLYPH_GENERATOR_PROTOCOL.GenerateGlyph()**

**Summary**

The function generates the glyph information according to the given font information. This function returns the glyph block in EFI_HII_GIBT_GLYPH_VARIABILITY type.
Protocol

typedef

EFI_STATUS
(EIFI_API *EFI_HII_GENERATE_GLYPH)(
  IN CONST EFI_HII_FONT_GLYPH_GENERATOR_PROTOCOL *This,
  IN CHAR16 Char,
  IN CONST EFI_FONT_DISPLAY_INFO *FontInfo,
  OUT EFI_HII_GIBT_VARIABILITY_BLOCK *GlyphBlock
);

Parameters

This EFI_HII_FONT_GLYPH_GENERATOR_PROTOCOL instance.
Char Character to retrieve.
FontInfo Font display information of this character.
GlyphBlock Pointer to retrieve the EFI_HII_GIBT_VARIABILITY_BLOCK

Description

This function generates the glyph information of the character in the specific font family. EFI_HII_GIBT_VARIABILITY_BLOCK is returned to GlyphBlock if GlyphBlock is not NULL. GlyphBlock can be called by EFI_HII_FONT_EX_PROTOCOL to retrieve the glyph information which are provided by the font family specific driver, or can be used to build up the HII font package if the HII font package with the specific font family does not exist in the HII database.

Status Codes Returned

<table>
<thead>
<tr>
<th>Code</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>EFI_SUCCESS</td>
<td>The glyph information was returned to GlyphBlock.</td>
</tr>
<tr>
<td>EFI_INVALID_PARAMETER</td>
<td>The FontInfo or GlyphBlock was NULL.</td>
</tr>
<tr>
<td>EFI_OUT_OF_RESOURCE</td>
<td>Not enough memory to generate the glyph information.</td>
</tr>
<tr>
<td>EFI_UNSUPPORTED</td>
<td>The font glyph generator can't generate the glyph for the given Char. This may caused by the unsupported character, font name font style or font size.</td>
</tr>
</tbody>
</table>

EFI_HII_FONT_GLYPH_GENERATOR_PROTOCOL.GenerateGlyphImage()

Summary

The function generates the glyph image according to the given font information. This function returns EFI_GRAPHICS_OUTPUT_BLT_PIXEL points to the EFI_IMAGE_OUTPUT buffer. This function is used for glyphs which are reported in the font database as EFI_HII_GIBT_GLYPH_VARIABILITY glyph blocks.
Protocol

typedef

EFI_STATUS

(EFIAPI *EFI_HII_GENERATE_GLYPH_IMAGE)(
    IN CONST EFI_HII_FONT_GLYPH_GENERATOR_PROTOCOL *This,
    IN CONST EFI_HII_GLYPH_INFO *Cell,
    IN UINT8 *GlyphBuffer,
    IN CONST EFI_FONT_DISPLAY_INFO *FontInfo,
    IN OUT EFI_IMAGE_OUTPUT *Image,
    IN INT32 *BltX,
    IN INT32 *BltY,
    IN BOOLEAN Transparent
);

Parameters

This       EFI_HII_FONT_GLYPH_GENERATOR_PROTOCOL instance.
Cell       Pointer to EFI_HII_GLYPH_INFO
GlyphBuffer The buffer points to the bitmap of glyph. This pointer points to
             GlyphBlock.BitmapData which returned from
             GenerateGlyph()function
FontInfo   Font display information of this glyph.
Image      Image output buffer to retrieve the glyph image.
BltX       Together with BltY, specifies the offset from the left and top
             edge of the image of the first character cell in the *Image.
BltY       Together with BltX, specifies the offset from the left and top
             edge of the image of the first character cell in the *Image.
Transparent If TRUE, the Background color is ignored and all"off" pixels in
             the character's drawn will use the pixel value from *Image.

Description

This function generates the glyph image of the character in the specific font family on the given
EFI_IMAGE_OUTPUT

Status Codes Returned

<table>
<thead>
<tr>
<th>Status Code</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>EFI_SUCCESS</td>
<td>The glyph image was generated in Image.</td>
</tr>
<tr>
<td>EFI_OUT_OF_RESOURCE</td>
<td>Not enough memory to generate image of the given glyph.</td>
</tr>
</tbody>
</table>
| EFI_UNSUPPORTED              | The font glyph generator can't generate the glyph for the given FontInfo. This may
                                caused by the unsupported font name, font style or font size. |
| EFI_INVALID_PARAMETER        | One or more parameters of Cell, GlyphBuffer, FontInfo, Image, BltX or BltY was NULL. |
34.8 Database Protocol

**EFI_HII_DATABASE_PROTOCOL**

**Summary**
Database manager for HII-related data structures.

**GUID**

```
#define EFI_HII_DATABASE_PROTOCOL_GUID \
{ 0xef9fc172, 0xa1b2, 0x4693,\ 
  { 0xb3, 0x27, 0x6d, 0x32, 0xfc, 0x41, 0x60, 0x42 }}
```

**Protocol**

```c
typedef struct _EFI_HII_DATABASE_PROTOCOL {
  EFI_HII_DATABASE_NEW_PACK    NewPackageList;
  EFI_HII_DATABASE_REMOVE_PACK RemovePackageList;
  EFI_HII_DATABASE_UPDATE_PACK UpdatePackageList;
  EFI_HII_DATABASE_LIST_PACKS ListPackageLists;
  EFI_HII_DATABASE_EXPORT_PACKS ExportPackageLists;
  EFI_HII_DATABASE_REGISTER_NOTIFY RegisterPackageNotify;
  EFI_HII_DATABASE_UNREGISTER_NOTIFY UnregisterPackageNotify;
  EFI_HII_FIND_KEYBOARD_LAYOUTS FindKeyboardLayouts;
  EFI_HII_GET_KEYBOARD_LAYOUT GetKeyboardLayout;
  EFI_HII_SET_KEYBOARD_LAYOUT SetKeyboardLayout;
  EFI_HII_DATABASE_GET_PACK_HANDLE GetPackageListHandle;
} EFI_HII_DATABASE_PROTOCOL;
```

**Members**

- **NewPackageList**
  Add a new package list to the HII database.

- **RemovePackageList**
  Remove a package list from the HII database.

- **UpdatePackageList**
  Update a package list in the HII database.

- **ListPackageLists**
  List the handles of the package lists within the HII database.

- **ExportPackageLists**
  Export package lists from the HII database.

- **RegisterPackageNotify**
  Register notification when packages of a certain type are installed.

- **UnregisterPackageNotify**
  Unregister notification of packages.
FindKeyboardLayout

Retrieves a list of the keyboard layouts in the system.

GetKeyboardLayout

Allows a program to extract the current keyboard layout. See the GetKeyboardLayout() function description.

SetKeyboardLayout

Changes the current keyboard layout. See the SetKeyboardLayout() function description.

GetPackageListHandle

Return the EFI handle associated with a given package list.

EFI_HII_DATABASE_PROTOCOL.NewPackageList()

Summary

Adds the packages in the package list to the HII database.

Prototype

typedef
EFI_STATUS
(EIFIAPI *EFI_HII_DATABASE_NEW_PACK) (   
   IN CONST EFI_HII_DATABASE_PROTOCOL *This,   
   IN CONST EFI_HII_PACKAGE_LIST_HEADER *PackageList,   
   IN CONST EFI_HANDLE DriverHandle, OPTIONAL   
   OUT EFI_HII_HANDLE *Handle
);

Parameters

This

A pointer to the EFI_HII_DATABASE_PROTOCOL instance.

PackageList

A pointer to an EFI_HII_PACKAGE_LIST_HEADER structure.

DriverHandle

Associate the package list with this EFI handle

Handle

A pointer to the EFI_HII_HANDLE instance. Type EFI_HII_HANDLE is defined in "Related Definitions" below.

Description

This function adds the packages in the package list to the database and returns a handle. If there is a EFI_DEVICE_PATH_PROTOCOL associated with the DriverHandle, then this function will create a package of type EFI_PACKAGE_TYPE_DEVICE_PATH and add it to the package list.
For each package in the package list, registered functions with the notification type \texttt{NEW\_PACK} and having the same package type will be called.

For each call to \texttt{NewPackageList()}, there should be a corresponding call to \texttt{EFI\_HII\_DATABASE\_PROTOCOL.RemovePackageList()}. 

\textbf{Related Definitions} 
\begin{verbatim}
typedef VOID *EFI\_HII\_HANDLE;
\end{verbatim}

\textbf{Status Codes Returns} 
\begin{itemize}
\item \texttt{EFI\_SUCCESS} \hspace{1cm} The package list associated with the \texttt{Handle} was added to the HII database.
\item \texttt{EFI\_OUT\_OF\_RESOURCES} \hspace{1cm} Unable to allocate necessary resources for the new database contents.
\item \texttt{EFI\_INVALID\_PARAMETER} \hspace{1cm} \texttt{PackageList} is NULL or \texttt{Handle} is \texttt{NULL}.
\end{itemize}

\texttt{EFI\_HII\_DATABASE\_PROTOCOL.RemovePackageList()} 

\textbf{Summary} 
Removes a package list from the HII database.

\textbf{Prototype} 
\begin{verbatim}
typedef EFI\_STATUS
(EIFIAPI *EFI\_HII\_DATABASE\_REMOVE\_PACK) (
  IN CONST EFI\_HII\_DATABASE\_PROTOCOL *This,
  IN EFI\_HII\_HANDLE Handle
);
\end{verbatim}

\textbf{Parameters} 
\begin{itemize}
\item \textit{This} \hspace{1cm} A pointer to the \texttt{EFI\_HII\_DATABASE\_PROTOCOL} instance.
\item \textit{Handle} \hspace{1cm} The handle that was registered to the data that is requested for removal. Type \texttt{EFI\_HII\_HANDLE} is defined in \texttt{EFI\_HII\_DATABASE\_PROTOCOL.NewPackageList()} in the Packages section.
\end{itemize}

\textbf{Description} 
This function removes the package list that is associated with a handle \texttt{Handle} from the HII database. Before removing the package, any registered functions with the notification type \texttt{REMOVE\_PACK} and the same package type will be called.

For each call to \texttt{EFI\_HII\_DATABASE\_PROTOCOL.NewPackageList()}, there should be a corresponding call to \texttt{RemovePackageList()}. 

Status Codes Returned

- **EFI_SUCCESS**: The data associated with the Handle was removed from the HII database.
- **EFI_NOT_FOUND**: The specified Handle is not in the Database.

### EFI_HII_DATABASE_PROTOCOL.UpdatePackageList()

**Summary**
Update a package list in the HII database.

**Prototype**
```c
typedef
EFI_STATUS
(EFIAPI *EFI_HII_DATABASE_UPDATE_PACK) (    
  IN CONST EFI_HII_DATABASE_PROTOCOL *This,    
  IN EFI_HII_HANDLE Handle,    
  IN CONST EFI_HII_PACKAGE_LIST_HEADER *PackageList,
);
```

**Parameters**
- **This**: A pointer to the EFI_HII_DATABASE_PROTOCOL instance.
- **Handle**: The handle that was registered to the data that is requested to be updated. Type EFI_HII_HANDLE is defined in EFI_HII_DATABASE_PROTOCOL.NewPackageList() in the Packages section.
- **PackageList**: A pointer to an instance of EFI_HII_PACKAGE_LIST_HEADER.

**Description**
This function updates the existing package list (which has the specified Handle) in the HII databases, using the new package list specified by PackageList. The update process has the following steps:

Collect all the package types in the package list specified by PackageList. A package type consists of the Type field of EFI_HII_PACKAGE_HEADER and, if the Type is EFI_HII_PACKAGE_TYPE_GUID, the Guid field, as defined in EFI_HII_GUID_PACKAGE_HDR.

Iterate through the packages within the existing package list in the HII database specified by Handle. If a package’s type matches one of the types collected in step 1, then perform the following steps:

- Call any functions registered with the notification type REMOVE_PACK.
- Remove the package from the package list and the HII database.

Add all of the packages within the new package list specified by PackageList, using the following steps:

- Add the package to the package list and the HII database.
Call any functions registered with the notification type **ADD_PACK**.

**Status Codes Returned**

- **EFI_SUCCESS**
  The HII database was successfully updated.
- **EFI_OUT_OF_RESOURCES**
  Unable to allocate enough memory for the updated database.
- **EFI_INVALID_PARAMETER**
  PackageList was **NULL**.
- **EFI_NOT_FOUND**
  The specified Handle is not in the Database.

**EFI_HII_DATABASE_PROTOCOL.ListPackageLists()**

**Summary**
Determines the handles that are currently active in the database.

**Prototype**

```c
typedef
EFI_STATUS
(EFIAPI *EFI_HII_DATABASE_LIST_PACKS) (  
  IN CONST EFI_HII_DATABASE_PROTOCOL *This,
  IN UINT8 PackageType,
  IN CONST EFI_GUID *PackageGuid,
  IN OUT UINTN *HandleBufferLength,
  OUT EFI_HII_HANDLE *Handle
);
```

**Parameters**

**This**
A pointer to the **EFI_HII_DATABASE_PROTOCOL** instance.

**PackageType**
Specifies the package type of the packages to list or **EFI_HII_PACKAGE_TYPE_ALL** for all packages to be listed.

**PackageGuid**
If PackageType is **EFI_HII_PACKAGE_TYPE_GUID**, then this is the pointer to the GUID which must match the Guid field of **EFI_HII_GUID_PACKAGE_HDR**. Otherwise, it must be NULL.

**HandleBufferLength**
On input, a pointer to the length of the handle buffer. On output, the length of the handle buffer that is required for the handles found.

**Handle**
An array of **EFI_HII_HANDLE** instances returned. Type **EFI_HII_HANDLE** is defined in **EFI_HII_DATABASE_PROTOCOL.NewPackageList()** in the Packages section.
Description
This function returns a list of the package handles of the specified type that are currently active in the database. The pseudo-type \texttt{EFI\_HII\_PACKAGE\_TYPE\_ALL} will cause all package handles to be listed.

Status Codes Returned

<table>
<thead>
<tr>
<th>Code</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>EFI_SUCCESS</td>
<td>A list of Packages was placed in \texttt{Handle} successfully. \texttt{HandleBufferLength} is updated with the actual length.</td>
</tr>
<tr>
<td>EFI_BUFFER_TOO_SMALL</td>
<td>The \texttt{HandleBufferLength} parameter indicates that \texttt{Handle} is too small to support the number of handles. \texttt{HandleBufferLength} is updated with a value that will enable the data to fit.</td>
</tr>
<tr>
<td>EFI_INVALID_PARAMETER</td>
<td>\texttt{HandleBufferLength} was \texttt{NULL}.</td>
</tr>
<tr>
<td>EFI_INVALID_PARAMETER</td>
<td>The value referenced by \texttt{HandleBufferLength} was not zero and \texttt{Handle} was NULL.</td>
</tr>
<tr>
<td>EFI_INVALID_PARAMETER</td>
<td>\texttt{PackageType} is a \texttt{EFI_HII_PACKAGE_TYPE_GUID} but \texttt{PackageGuid} is not \texttt{NULL}.</td>
</tr>
<tr>
<td>EFI_INVALID_PARAMETER</td>
<td>\texttt{PackageType} is a \texttt{EFI_HII_PACKAGE_TYPE_GUID} but \texttt{PackageGuid} is \texttt{NULL}.</td>
</tr>
<tr>
<td>EFI_NOT_FOUND</td>
<td>No matching handles were found</td>
</tr>
</tbody>
</table>

\textbf{EFI\_HII\_DATABASE\_PROTOCOL.ExportPackageLists()}

Summary
Exports the contents of one or all package lists in the HII database into a buffer.

Prototype

```c
typedef EFI\_STATUS
\( (\text{EFIAPI} *\text{EFI\_HII\_DATABASE\_EXPORT\_PKCS}) (\)
\( \text{IN \text{CONST}} \text{EFI\_HII\_DATABASE\_PROTOCOL} *\text{This}, \)
\( \text{IN \text{EFI\_HII\_HANDLE}} \quad \text{Handle}, \)
\( \text{IN \text{OUT}} \text{UINTN} \quad *\text{BufferSize}, \)
\( \text{OUT \text{EFI\_HII\_PACKAGE\_LIST\_HEADER}} \quad *\text{Buffer} \)
\) ;
```

Parameters

\texttt{This}  
A pointer to the \texttt{EFI\_HII\_DATABASE\_PROTOCOL} instance.

\texttt{Handle}  
An \texttt{EFI\_HII\_HANDLE} that corresponds to the desired package list in the HII database to export or NULL to indicate all package lists should be exported.
BufferSize
On input, a pointer to the length of the buffer. On output, the length of the buffer that is required for the exported data.

Buffer
A pointer to a buffer that will contain the results of the export function.

Description
This function will export one or all package lists in the database to a buffer. For each package list exported, this function will call functions registered with `EXPORT_PACK` and then copy the package list to the buffer. The registered functions may call `EFI_HII_DATABASE_PROTOCOL.UpdatePackageList()` to modify the package list before it is copied to the buffer.

If the specified `BufferSize` is too small, then the status `EFI_BUFFER_TOO_SMALL` will be returned and the actual package size will be returned in `Buffer`. 

Status Codes Returned

<table>
<thead>
<tr>
<th>Status Code</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>EFI_SUCCESS</td>
<td>Package exported.</td>
</tr>
<tr>
<td>EFI_BUFFER_TOO_SMALL</td>
<td>BufferSize is too small to hold the package.</td>
</tr>
<tr>
<td>EFI_INVALID_PARAMETER</td>
<td>BufferSize was NULL</td>
</tr>
<tr>
<td>EFI_INVALID_PARAMETER</td>
<td>The value referenced by BufferSize was not zero and Buffer was NULL.</td>
</tr>
<tr>
<td>EFI_NOT_FOUND</td>
<td>The specified Handle could not be found in the current database.</td>
</tr>
</tbody>
</table>

`EFI_HII_DATABASE_PROTOCOL.RegisterPackageNotify()`

Summary
Registers a notification function for HII database-related events.

Prototype

```c
typedef
EFI_STATUS
(EIFIAPI *EFI_HII_DATABASE_REGISTER_NOTIFY) (  
    IN CONST EFI_HII_DATABASE_PROTOCOL *This,  
    IN UINT8 PackageType,  
    IN CONST EFI_GUID *PackageGuid,  
    IN CONST EFI_HII_DATABASE_NOTIFY PackageNotifyFn,  
    IN EFI_HII_DATABASE_NOTIFY_TYPE NotifyType,  
    OUT EFI_HANDLE *NotifyHandle  
);  
```

Parameters

`This`
A pointer to the `EFI_HII_DATABASE_PROTOCOL` instance.
**PackageType**

The package type. See `EFI_HII_PACKAGE_TYPE_x` in `EFI_HII_PACKAGE_HEADER`.

**PackageGuid**

If `PackageType` is `EFI_HII_PACKAGE_TYPE_GUID`, then this is the pointer to the GUID which must match the `Guid` field of `EFI_HII_GUID_PACKAGE_HDR`. Otherwise, it must be NULL.

**PackageNotifyFn**

Points to the function to be called when the event specified by `NotificationType` occurs. See `EFI_HII_DATABASE_NOTIFY`.

**NotifyType**

Describes the types of notification which this function will be receiving. See `EFI_HII_DATABASE_NOTIFY_TYPE` for more a list of types.

**NotifyHandle**

Points to the unique handle assigned to the registered notification. Can be used in `EFI_HII_DATABASE_PROTOCOL.UnregisterPackageNotify()` to stop notifications.

**Description**

This function registers a function which will be called when specified actions related to packages of the specified type occur in the HII database. By registering a function, other HII-related drivers are notified when specific package types are added, removed or updated in the HII database.

Each driver or application which registers a notification should use `EFI_HII_DATABASE_PROTOCOL.UnregisterPackageNotify()` before exiting.

If a driver registers a `NULL PackageGuid` when `PackageType` is `EFI_HII_PACKAGE_TYPE_GUID`, a notification will occur for every package of type `EFI_HII_PACKAGE_TYPE_GUID` that is registered.

**Related Definitions**

- `EFI_HII_PACKAGE_HEADER` is defined in `EFI_HII_PACKAGE_HEADER`.
- `EFI_HII_DATABASE_NOTIFY` is defined in `EFI_HII_DATABASE_NOTIFY`.
- `EIF_HII_DATABASE_NOTIFY_TYPE` is defined in `EFI_HII_DATABASE_NOTIFY_TYPE`. 
Returned Status Codes

EFI_SUCCESS  
Notification registered successfully.

EFI_OUT_OF_RESOURCES  
Unable to allocate necessary data structures.

EFI_INVALID_PARAMETER  
NotifyHandle is NULL.

EFI_INVALID_PARAMETER  
PackageType is not a EFI_HII_PACKAGE_TYPE_GUID but PackageGuid is not NULL.

EFI_INVALID_PARAMETER  
PackageType is a EFI_HII_PACKAGE_TYPE_GUID but PackageGuid is NULL.

EFI_HII_DATABASE_PROTOCOL.UnregisterPackageNotify()

Summary
Removes the specified HII database package-related notification.

Prototype

typedef  
EFI_STATUS  
(EFIAPI *EFI_HII_DATABASE_UNREGISTER_NOTIFY) (  
    IN CONST EFI_HII_DATABASE_PROTOCOL *This,  
    IN EFI_HANDLE NotificationHandle  
);

Parameters

This
A pointer to the EFI_HII_DATABASE_PROTOCOL instance.

NotificationHandle
The handle of the notification function being unregistered.

Returned Status Codes

EFI_SUCCESS  
Invalidated

EFI_NOT_FOUND  
The NotificationHandle could not be found in the database.

EFI_HII_DATABASE_PROTOCOL.FindKeyboardLayouts()

Summary
Retrieves a list of the keyboard layouts in the system.
Prototype

```c
typedef EFI_STATUS (EFIAPI *EFI_HII_FIND_KEYBOARD_LAYOUTS) (IN CONST EFI_HII_DATABASE_PROTOCOL *This,
   IN OUT UINT16    *KeyGuidBufferLength,
   OUT EFI_GUID     *KeyGuidBuffer);
```

Parameters

- **This**
  A pointer to the **EFI_HII_DATABASE_PROTOCOL** instance.

- **KeyGuidBufferLength**
  On input, a pointer to the length of the keyboard GUID buffer. On output, the length of the handle buffer that is required for the handles found.

- **KeyGuidBuffer**
  An array of keyboard layout GUID instances returned.

Description

This routine retrieves an array of GUID values for each keyboard layout that was previously registered in the system.

Status Codes Returned

- **EFI_SUCCESS**
  KeyGuidBuffer was updated successfully.

- **EFI_BUFFER_TOO_SMALL**
  The **KeyGuidBufferLength** parameter indicates that **KeyGuidBuffer** is too small to support the number of GUIDs. **KeyGuidBufferLength** is updated with a value that will enable the data to fit.

- **EFI_INVALID_PARAMETER**
  **KeyGuidBufferLength** is **NULL**.

- **EFI_INVALID_PARAMETER**
  The value referenced by **KeyGuidBufferLength** is not zero and **KeyGuidBuffer** is **NULL**.

**EFI_HII_DATABASE_PROTOCOL.GetKeyboardLayout()**

Summary

Retrieves the requested keyboard layout.
Prototype

Prototype

```c
typedef
EFI_STATUS
(EFI_API *EFI_HII_GET_KEYBOARD_LAYOUT) (
    IN CONST EFI_HII_DATABASE_PROTOCOL *This,
    IN EFI_GUID *KeyGuid,
    IN_OUT UINT16 *KeyboardLayoutLength,
    OUT EFI_HII_KEYBOARD_LAYOUT *KeyboardLayout
);
```

Parameters

This

A pointer to the `EFI_HII_DATABASE_PROTOCOL` instance.

KeyGuid

A pointer to the unique ID associated with a given keyboard layout. If `KeyGuid` is `NULL` then the current layout will be retrieved.

KeyboardLayout

A pointer to a buffer containing the retrieved keyboard layout. below.

KeyboardLayoutLength

On input, a pointer to the length of the KeyboardLayout buffer. On output, the length of the data placed into KeyboardLayout.

Description

This routine retrieves the requested keyboard layout. The layout is a physical description of the keys on a keyboard and the character(s) that are associated with a particular set of key strokes.

Related Definitions

```c
//********************************************************************************
// EFI_HII_KEYBOARD_LAYOUT
//********************************************************************************
typedef struct {
    UINT16 LayoutLength;
    EFI_GUID Guid;
    UINT32 LayoutDescriptorStringOffset;
    UINT8 DescriptorCount;
    EFI_KEY_DESCRIPTOR Descriptors[];
} EFI_HII_KEYBOARD_LAYOUT;
```

`LayoutLength`

The length of the current keyboard layout.

`Guid`

The unique ID associated with this keyboard layout.
**LayoutDescriptorStringOffset**

An offset location (0 is the beginning of the `EFI_KEYBOARD_LAYOUT` instance) of the string which describes this keyboard layout. The data that is being referenced is in `EFI_DESCRIPTION_STRING_BUNDLE` format.

**DescriptorCount**

The number of Descriptor entries in this layout.

**Descriptors**

An array of key descriptors.

```c
//**********************************************************
// EFI_DESCRIPTION_STRING - byte packed data
//**********************************************************
CHAR16  Language[];
CHAR16   Space;
//CHAR16  DescriptionString[];

Language

The language in RFC 4646 format to associate with DescriptionString.

**Space**

A space (U-0x0020) character to force as a separator between the Language field and the formal description string.

**DescriptionString**

A null-terminated description string.

```c
//**********************************************************
// EFI_DESCRIPTION_STRING_BUNDLE - byte packed data
// Example: 2en-US English Keyboard<null>es-ES Keyboard en ingles<null>
//<null> = U-0000
//**********************************************************
UINT16   DescriptionCount;
EFI_DESCRIPTION_STRING   DescriptionString[];

DescriptionCount

The number of description strings.

**DescriptionString**

An array of language-specific description strings.
typedef struct {
  EFI_KEY    Key;
  CHAR16    Unicode;
  CHAR16    ShiftedUnicode;
  CHAR16    AltGrUnicode;
  CHAR16    ShiftedAltGrUnicode;
  UINT16    Modifier;
  UINT16    AffectedAttribute;
} EFI_KEY_DESCRIPTOR;

// A key which is affected by all the standard shift modifiers.  
// Most keys would be expected to have this bit active.  
#define EFI_AFFECTED_BY_STANDARD_SHIFT 0x0001

// This key is affected by the caps lock so that if a keyboard  
// driver would need to disambiguate between a key which had a  
// "1" defined versus a "a" character. Having this bit turned on  
// would tell the keyboard driver to use the appropriate shifted // state or  
not.  
#define EFI_AFFECTED_BY_CAPS_LOCK 0x0002

// Similar to the case of CAPS lock, if this bit is active, the  
// key is affected by the num lock being turned on.  
#define EFI_AFFECTED_BY_NUM_LOCK 0x0004

Key
Used to describe a physical key on a keyboard. Type EFI_KEY is defined below.

Unicode
Unicode character code for the Key.

ShiftedUnicode
Unicode character code for the key with the shift key being held down.

AltGrUnicode
Unicode character code for the key with the Alt-GR being held down.

ShiftedAltGrUnicode
Unicode character code for the key with the Alt-GR and shift keys being held down.

Modifier
Modifier keys are defined to allow for special functionality that is not necessarily accomplished by a printable character. Many of these modifier keys are flags to
toggle certain state bits on and off inside of a keyboard driver. Values for Modifier are defined below.

```c
//***************************************************
// EFI_KEY
//************************************************************************
typedef enum {
  EfiKeyLCtrl, EfiKeyA0, EfiKeyLAlt, EfiKeySpaceBar, EfiKeyA2, EfiKeyA3,
  EfiKeyA4, EfiKeyRCtrl, EfiKeyLeftArrow, EfiKeyDownArrow, EfiKeyRightArrow,
  EfiKeyZero, EfiKeyPeriod, EfiKeyEnter, EfiKeyLShift, EfiKeyB0, EfiKeyB1,
  EfiKeyB2, EfiKeyB3, EfiKeyB4, EfiKeyB5, EfiKeyB6, EfiKeyB7, EfiKeyB8,
  EfiKeyB9, EfiKeyB10, EfiKeyRShift, EfiKeyUpArrow, EfiKeyOne, EfiKeyTwo,
  EfiKeyThree, EfiKeyCapsLock, EfiKeyC1, EfiKeyC2, EfiKeyC3, EfiKeyC4,
  EfiKeyC5, EfiKeyC6, EfiKeyC7, EfiKeyC8, EfiKeyC9, EfiKeyC10, EfiKeyC11,
  EfiKeyC12, EfiKeyFour, EfiKeyFive, EfiKeySix, EfiKeyPlus, EfiKeyTab,
  EfiKeyD1, EfiKeyD2, EfiKeyD3, EfiKeyD4, EfiKeyD5, EfiKeyD6, EfiKeyD7,
  EfiKeyD8, EfiKeyD9, EfiKeyD10, EfiKeyD11, EfiKeyD12, EfiKeyD13, EfiKeyDel,
  EfiKeyEnd, EfiKeyPgDn, EfiKeySeven, EfiKeyEight, EfiKeyNine, EfiKeyE0,
  EfiKeyE1, EfiKeyE2, EfiKeyE3, EfiKeyE4, EfiKeyE5, EfiKeyE6, EfiKeyE7,
  EfiKeyE8, EfiKeyE9, EfiKeyE10, EfiKeyE11, EfiKeyE12, EfiKeyBackSpace,
  EfiKeyIns, EfiKeyHome, EfiKeyPgUp, EfiKeyNlck, EfiKeySlash, EfiKeyAsterisk,
  EfiKeyMinus, EfiKeyEsc, EfiKeyF1, EfiKeyF2, EfiKeyF3, EfiKeyF4, EfiKeyF5,
  EfiKeyF6, EfiKeyF7, EfiKeyF8, EfiKeyF9, EfiKeyF10, EfiKeyF11, EfiKeyF12,
  EfiKeyPrint, EfiKeySlck, EfiKeyPause, EfiKeyIntl0, EfiKeyIntl1,
  EfiKeyIntl2, EfiKeyIntl3, EfiKeyIntl4, EfiKeyIntl5, EfiKeyIntl6,
  EfiKeyIntl7, EfiKeyIntl8, EfiKeyIntl9
} EFI_KEY;
```

See the figure below for which key corresponds to the values in the enumeration above. For example, **EfiKeyLCtrl** corresponds to the left control key in the lower-left corner of the keyboard, **EfiKeyFour** corresponds to the 4 key on the numeric keypad, and **EfiKeySLck** corresponds to the Scroll Lock key in the upper-right corner of the keyboard.

![Keyboard Layout](image-url)

**Figure 34-3 Keyboard Layout**
// Modifier values
//******************************************************************************
#define EFI_NULL_MODIFIER         0x0000
#define EFI_LEFT_CONTROL_MODIFIER     0x0001
#define EFI_RIGHT_CONTROL_MODIFIER    0x0002
#define EFI_LEFT_ALT_MODIFIER       0x0003
#define EFI_RIGHT_ALT_MODIFIER      0x0004
#define EFI_ALT_GR_MODIFIER        0x0005
#define EFI_INSERT_MODIFIER        0x0006
#define EFI_DELETE_MODIFIER        0x0007
#define EFI_PAGE_DOWN_MODIFIER      0x0008
#define EFI_PAGE_UP_MODIFIER       0x0009
#define EFI_HOME_MODIFIER         0x000A
#define EFI_END_MODIFIER         0x000B
#define EFI_LEFT_SHIFT_MODIFIER      0x000C
#define EFI_RIGHT_SHIFT_MODIFIER     0x000D
#define EFI_CAPS_LOCK_MODIFIER      0x000E
#define EFI_NUM_LOCK_MODIFIER       0x000F
#define EFI_LEFT_ARROW_MODIFIER      0x0010
#define EFI_RIGHT_ARROW_MODIFIER     0x0011
#define EFI_DOWN_ARROW_MODIFIER      0x0012
#define EFI_UP_ARROW_MODIFIER       0x0013
#define EFI_NS_KEY_MODIFIER        0x0014
#define EFI_NS_KEY_DEPENDENCY_MODIFIER  0x0015
#define EFI_FUNCTION_KEY_ONE_MODIFIER   0x0016
#define EFI_FUNCTION_KEY_TWO_MODIFIER   0x0017
#define EFI_FUNCTION_KEY_THREE_MODIFIER  0x0018
#define EFI_FUNCTION_KEY_FOUR_MODIFIER  0x0019
#define EFI_FUNCTION_KEY_FIVE_MODIFIER  0x001A
#define EFI_FUNCTION_KEY_SIX_MODIFIER   0x001B
#define EFI_FUNCTION_KEY_SEVEN_MODIFIER  0x001C
#define EFI_FUNCTION_KEY_EIGHT_MODIFIER  0x001D
#define EFI_FUNCTION_KEY_NINE_MODIFIER  0x001E
#define EFI_FUNCTION_KEY_TEN_MODIFIER   0x001F
#define EFI_FUNCTION_KEY_ELEVEN_MODIFIER 0x0020
#define EFI_FUNCTION_KEY_TWELVE_MODIFIER 0x0021

// Keys that have multiple control functions based on modifier
// settings are handled in the keyboard driver implementation.
// For instance PRINT_KEY might have a modifier held down and
// is still a nonprinting character, but might have an alternate
// control function like SYSREQUEST
//
#define EFI_PRINT_MODIFIER        0x0022
#define EFI_SYS_REQUEST_MODIFIER     0x0023
#define EFI_SCROLL_LOCK_MODIFIER     0x0024
#define EFI_PAUSE_MODIFIER        0x0025
#define EFI_BREAK_MODIFIER        0x0026
#define EFI_LEFT_LOGO_MODIFIER      0x0027
#define EFI_RIGHT_LOGO_MODIFIER      0x0028
#define EFI_MENU_MODIFIER         0x0029

Status Codes Returned

- **EFI_SUCCESS**
  The keyboard layout was retrieved successfully.

- **EFI_NOT_FOUND**
  The requested keyboard layout was not found.

- **EFI_BUFFER_TOO_SMALL**
  The `KeyboardLayoutLength` parameter indicates the `KeyboardLayout` is too small to hold the keyboard layout.

- **EFI_INVALID_PARAMETER**
  `KeyboardLayoutLength` is **NULL**

- **EFI_INVALID_PARAMETER**
  The value referenced by `KeyboardLayoutLength` is not zero and `KeyboardLayout` is NULL.

### EFI_HII_DATABASE_PROTOCOL.SetKeyboardLayout()

**Summary**

Sets the currently active keyboard layout.

**Prototype**

```c
typedef
 EFI_STATUS
 (EFIAPI *EFI_HII_SET_KEYBOARD_LAYOUT) ( 
    IN CONST EFI_HII_DATABASE_PROTOCOL   *This, 
    IN EFI_GUID                           *KeyGuid 
);
```

**Parameters**

- **This**
  A pointer to the **EFI_HII_DATABASE_PROTOCOL** instance.

- **KeyGuid**
  A pointer to the unique ID associated with a given keyboard layout.

**Description**

This routine sets the default keyboard layout to the one referenced by **KeyGuid**. When this routine is called, an event will be signaled of the **EFI_HII_SET_KEYBOARD_LAYOUT_EVENT_GUID** group type. This is so that agents which are sensitive to the current keyboard layout being changed can be notified of this change.
Related Definitions

GUID

```c
#define EFI_HII_SET_KEYBOARD_LAYOUT_EVENT_GUID \
   { 0x14982a4f, 0xb0ed, 0x45b8, \
     { 0xa8, 0x11, 0x5a, 0x7a, 0x9b, 0xc2, 0x32, 0xdf }}
```

Status Codes Returned

- **EFI_SUCCESS** The current keyboard layout was successfully set.
- **EFI_NOT_FOUND** The referenced keyboard layout was not found, so action was taken.
- **EFI_INVALID_PARAMETER** 
  
  `KeyGuid` is NULL.

**EFI_HII_DATABASE_PROTOCOL.GetPackageListHandle()**

**Summary**

Return the EFI handle associated with a package list.

**Prototype**

```c
typedef EFI_STATUS 
(EFIAPI *EFI_HII_DATABASE_GET_PACK_HANDLE) ( 
   IN CONST EFI_HII_DATABASE_PROTOCOL *This, 
   IN EFI_HII_HANDLE PackageListHandle, 
   OUT EFI_HANDLE *DriverHandle 
);
```

**Parameters**

- **This**
  A pointer to the **EFI_HII_DATABASE_PROTOCOL** instance.

- **PackageListHandle**
  An **EFI_HII_HANDLE** that corresponds to the desired package list in the HII database.

- **DriverHandle**
  On return, contains the **EFI_HANDLE** which was registered with the package list in **NewPackageList()**.
Status Codes Returned

EFI_SUCCESS
The DriverHandle was returned successfully.
EFI_INVALID_PARAMETER
The PackageListHandle was not valid.
EFI_INVALID_PARAMETER
The DriverHandle must not be NULL.

34.8.1 Database Structures

EFI_HII_DATABASE_NOTIFY

Summary
Handle a registered notification for a package change to the database.

Prototype
typedef
EFI_STATUS
(EFI_API *EFI_HII_DATABASE_NOTIFY) (  
IN UINT8 PackageType,
IN CONST EFI_GUID *PackageGuid,
IN CONST EFI_HII_PACKAGE_HEADER *Package,
IN EFI_HII_HANDLE Handle,
IN EFI_HII_DATABASE_NOTIFY_TYPE NotifyType
);

Parameters

PackageType
Package type of the notification.

PackageGuid
If PackageType is EFI_HII_PACKAGE_TYPE_GUID, then this is the pointer to the
GUID from the Guid field of EFI_HII_GUID_PACKAGE_HDR. Otherwise, it must be
NULL.

Package
Points to the package referred to by the notification

Handle
The handle of the package list which contains the specified package.

NotifyType
The type of change concerning the database. See
EFI_HII_DATABASE_NOTIFY_TYPE.
Description

Functions which are registered to receive notification of database events have this prototype. The actual event is encoded in NotifyType. The following table describes how PackageType, PackageGuid, Handle, and Package are used for each of the notification types.

<table>
<thead>
<tr>
<th>Notification Type</th>
<th>Parameter Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>NEW_PACK</td>
<td>PackageType and PackageGuid are the type of the new package. Package points to the new package. Handle is the handle of the package list which is being added to the database.</td>
</tr>
<tr>
<td>REMOVE_PACK</td>
<td>PackageType and PackageGuid are the type of the package which is being removed. Package points to the package being removed. Handle is the package list from which the package is being removed.</td>
</tr>
<tr>
<td>EXPORT_PACK</td>
<td>PackageType and PackageGuid are the type of the package being exported. Package points to the existing package in the database. Handle is the package list being exported.</td>
</tr>
<tr>
<td>ADD_PACK</td>
<td>PackageType and PackageGuid are the type of the package being added. Package points to the package being added. Handle is the package list to which the package is being added.</td>
</tr>
</tbody>
</table>

EFI_HII_DATABASE_NOTIFY_TYPE

typedef UINTN EFI_HII_DATABASE_NOTIFY_TYPE;

#define EFI_HII_DATABASE_NOTIFY_NEW_PACK  0x00000001
#define EFI_HII_DATABASE_NOTIFY_REMOVE_PACK  0x00000002
#define EFI_HII_DATABASE_NOTIFY_EXPORT_PACK  0x00000004
#define EFI_HII_DATABASE_NOTIFY_ADD_PACK  0x00000008
35 - HII Configuration Processing and Browser Protocol

35.1 Introduction
This section describes the data and APIs used to manage the system’s configuration: the actual data that describes the knobs and settings.

35.1.1 Common Configuration Data Format
The configuration data is stored as name / value string pairs. As in e.g. HTML, the name and value are separated by ‘=’ and the pairs are separated one from the next by ‘&’. The configuration data structures are thus variable length UNICODE (UCS-2) strings.

Certain names and values have limitations on their syntax to manage routing and to enable extended support for common storage mechanisms.

35.1.2 Data Flow
There is a two-way flow through the hierarchy of drivers and protocols that parallels the flow in other parts of HII. Initially, the flow is from the drivers up to the HII database and on to configuration applications. When changes to configuration are accepted, the flow reverses itself, going from the configuration applications through the HII database protocols back to the drivers through separate protocols.

The flow from driver up consists of the current and alternative (default) configurations. The flow down from the configuration applications consists of changed configurations.

The protocol managed by the HII Database is known as the EFI HII Configuration Routing Protocol, while the one presented by the drivers themselves is known as the EFI HII Configuration Access Protocol. The HII Configuration Routing Protocol is the only one that outside callers should invoke.

35.2 Configuration Strings
The configuration strings follow the same general format as HTTP argument strings, which is to say ‘&’ separated name / value pairs. The name and value are separated by ‘=’. The strings are a subset of full HTML argument strings and do not require quoting, the ‘%’ character sequences used to insert spaces, ampersands, equal signs, and the like into HTTP argument strings.

35.2.1 String Syntax
Assumptions are typical for BNF with the following extensions
Characters in single quotes, e.g. ‘a’, indicate terminals.
Square brackets immediately followed by a number n indicate that the contents are to be repeated n times, so [‘a’]4 would be “aaaa”.
An italicized non-terminal, e. g. <All Printable ASCII Characters> is used to indicate a set of terminals whose definition is outside the scope of this document.

The syntax for configuration strings is as follows.
35.2.1.1 Basic forms

<Dec19> ::= '1' | '2' | ... | '9'
<DecCh> ::= '0' | <Dec19>
<HexAf> ::= 'a' | 'b' | 'c' | 'd' | 'e' | 'f'
<Hex1f> ::= <Dec19> | <HexAf>
<HexCh> ::= <DecCh> | <HexAf>
<Number> ::= <HexCh>+<Dec19>
<Alpha> ::= 'a' | ... | 'z' | 'A' | ... | 'Z'

35.2.1.2 Types

<Guid> ::= <HexCh>32
<LabelStart> ::= <Alpha> | "_"
<LabelBody> ::= <LabelStart> | <DecCh>
<label> ::= <LabelStart> [<LabelBody]*
<Char> ::= <HexCh>4
<String> ::= [<Char>]+
<AltCfgId> ::= <HexCh>4

35.2.1.3 Routing elements

<GuidHdr> ::= 'GUID='<Guid>
<NameHdr> ::= 'NAME='<String>
<PathHdr> ::= 'PATH='<UEFI binary Device Path represented as hex number>
<DescHdr> ::= 'ALTCFG=<AltCfgId>
<ConfigHdr> ::= <GuidHdr>'&'<NameHdr>'&'<PathHdr>
<AltConfigHdr> ::= <ConfigHdr> '&'<DescHdr>

35.2.1.4 Body elements

<ConfigBody> ::= <ConfigElement>*
<ConfigElement> ::= '&'<BlockConfig> | '&'<NvConfig>
<BlockName> ::= 'OFFSET='<Number>'&WIDTH='<Number>
<BlockConfig> ::= <BlockName>'&VALUE='<Number>
<RequestElement> ::= '&'<BlockName> | '&'<Label>
<NvConfig> ::= <Label>'='<String> | <Label>'='<Number>

35.2.1.5 Configuration strings

<ConfigRequest> ::= <ConfigHdr><RequestElement>*
<MultiConfigRequest> ::= <ConfigRequest>['&'<ConfigRequest>]*
<ConfigResp> ::= <ConfigHdr><ConfigBody>
<AltResp> ::= <AltConfigHdr><ConfigBody>
<ConfigAltResp> ::= <ConfigResp> ['&'<AltResp>]*
<MultiConfigAltResp> ::= <ConfigAltResp> ['&'<ConfigAltResp>]*
<MultiConfigResp> ::= <ConfigResp> ['&'<ConfigResp>]*

Notes:
The `<Number>` represents a data buffer and is encoded as a sequence of bytes in the format %02x in the same order as the buffer bytes reside in memory.

The `<Guid>` represents a hex encoding of GUID and is encoded as a sequence of bytes in the format %02x in the same order as the GUID bytes reside in memory.

The syntax for a `<Label>` is the C label (e.g. Variable) syntax.

The `<ConfigHdr>` provides routing information. The name field is required even if non-block storage is targeted. In these cases, it may be used as a way to distinguish like storages from one another when a driver is being used.

The `<BlockName>` provides addressing information for managing block (e.g. UEFI Variable) storage. The first number provides the byte offset into the block while the second provides the length of bytes.

The `<PathHdr>` presents a hex encoding of a UEFI device path. This is not the printable path since the printable path is optional in UEFI and to enable simpler comparisons. The data is encoded as strings with the format %02x bytes in the same order as the device path resides in RAM memory.

The `<ConfigRequest>` provides a mechanism to request the current configuration for one or more elements.

The `<AltCfgId>` is the identifier of a configuration declared in the corresponding IFR.

The name ‘GUID’ is also used to separate `<String>` or `<ConfigRequest>` elements in the equivalent Multi version. That is:

```
GUID=...&NAME=...&...&fred=12&GUID=...&NAME=...&...&goyle=11
```

Indicates two `<String>`, with one ending with fred=12.

The following are reserved `<name>`s and cannot be used as names in a `<ConfigElement>`:

- GUID
- NAME
- PATH
- ALTCFG
- OFFSET
- WIDTH
- VALUE
35.2.1.6 Keyword strings

<table>
<thead>
<tr>
<th>Tag</th>
<th>Syntax</th>
</tr>
</thead>
<tbody>
<tr>
<td>&lt;NamespaceId&gt;</td>
<td>::= 'NAMESPACE='&lt;String&gt;'&amp;'</td>
</tr>
<tr>
<td>&lt;Keyword&gt;</td>
<td>::= 'KEYWORD='&lt;String&gt;[':&lt;DecCh&gt;(1/4)]</td>
</tr>
<tr>
<td>&lt;DataFilter&gt;</td>
<td>::= 'Buffer'</td>
</tr>
<tr>
<td>&lt;UsageFilter&gt;</td>
<td>::= '&lt;ReadOnly'</td>
</tr>
<tr>
<td>&lt;Filter&gt;</td>
<td>::= &lt;UsageFilter&gt;</td>
</tr>
<tr>
<td>&lt;ValueRange&gt;</td>
<td>::= '&amp;MAX='&lt;Number&gt;['&amp;MIN='&lt;Number&gt;['&amp;STEP='&lt;Number&gt;]]</td>
</tr>
<tr>
<td>&lt;ValueOption&gt;</td>
<td>::= '&amp;OPTIONVALUE='&lt;Number&gt;['&amp;OPTIONSTRING='&lt;String&gt;</td>
</tr>
<tr>
<td></td>
<td>['&amp;VALUETYPE='Numeric'</td>
</tr>
<tr>
<td>&lt;ValueAttribute&gt;</td>
<td>::= [&lt;ValueRange&gt;][&lt;ValueOption&gt;*]</td>
</tr>
<tr>
<td>&lt;Default&gt;</td>
<td>::= ['&amp;STANDARDDEFAULT='&lt;Number&gt;['&amp;MFGDEFAULT='&lt;Number&gt;['&amp;SAFEDEFAULT='&lt;Number&gt;]]</td>
</tr>
<tr>
<td>&lt;Display&gt;</td>
<td>::= '&amp;DISPLAYNAME='&lt;String&gt;</td>
</tr>
<tr>
<td>&lt;DataType&gt;</td>
<td>::= '&amp;DATATYPE='&lt;DataFilter&gt;</td>
</tr>
<tr>
<td>&lt;KeywordInfoFilter&gt;</td>
<td>::= 'All'[['DataType'][['ValueAttribute']][['Default'] ['DisplayName'] ]</td>
</tr>
<tr>
<td>&lt;Boolean&gt;</td>
<td>::= 'true'</td>
</tr>
<tr>
<td>&lt;KeywordInfoRequest&gt;</td>
<td>::= 'KEYWORDINFO='&lt;KeywordInfoFilter&gt;</td>
</tr>
<tr>
<td>&lt;KeywordInfoResp&gt;</td>
<td>::= [&lt;DataType&gt;][&lt;ValueAttribute&gt;][&lt;Default&gt;][&lt;Display&gt;]</td>
</tr>
<tr>
<td>&lt;KeywordRequest&gt;</td>
<td>::= [&lt;PathHdr]&gt;['&amp;']&lt;Keyword&gt; ['&amp;&lt;KeywordInfoRequest&gt;['&amp;&lt;Filter&gt;]</td>
</tr>
<tr>
<td>&lt;KeywordResp&gt;</td>
<td>::= &lt;NamespaceId&gt;&lt;PathHdr&gt;['&amp;']&lt;Keyword&gt;['&amp;VALUE='&lt;Number&gt; ['&amp;READONLY']&lt;KeywordInfoResp&gt; ]</td>
</tr>
<tr>
<td>&lt;MultiKeywordRequest&gt;</td>
<td>::= &lt;KeywordRequest&gt;['&amp;&lt;KeywordRequest&gt;*]</td>
</tr>
<tr>
<td>&lt;MultiKeywordResp&gt;</td>
<td>::= &lt;KeywordResp&gt;['&amp;&lt;KeywordResp&gt;*]</td>
</tr>
</tbody>
</table>

**Note:** For Keyword definitions, see the UEFI Configuration Namespace Registry document on [http://uefi.org/uefi](http://uefi.org/uefi).
The `<NameSpaceId>` element is equivalent to the platform configuration language being used for the keyword definition.

The `<Keyword>` element uses the ‘KEYWORD=’ name to designate that immediately following the reserved name is a string value associated with a configuration namespace keyword as defined in the Configuration NameSpace Registry document (http://uefi.org/uefi).

Typically, when a Keyword is defined, the value is a solitary string such as “BIOSVendor”. However, when certain Keywords are intended to represent a setting that may have multiple instances (e.g. ChipsetSATAPortEnable), that is when a “<DecCh>({1/4})” suffix will be appended to the keyword definition. In that case, we might see something like: “ChipsetSATAPortEnable:5” if a particular platform had at least five SATA ports and one of the questions was represented by the aforementioned string. It would also be reasonable to expect that there might also be a “ChipsetSATAPortEnable:1” and a “:2”, “:3” etc.

If the `<PathHdr>` element within `<KeywordRequest>` is omitted, then all instances are returned.

If the Keyboard Handler protocol knows or detects that a particular Keyword is read-only, then the `<KeywordResp>` must include the “&READONLY” tag.

The `<DataFilter>` element specifies the optional filter based on data type to use when a request is made. If no filtering is desired, then this element must be omitted from the `<KeywordRequest>`. Filtering is not guaranteed to work on any platform configuration language that isn’t defined in the UEFI Configuration Namespace Document.

**DataFilter.Buffer**

HII questions with EFI_IFR_TYPE_BUFFER type are treated as this type. This is most commonly represented in ‘C’ as a VOID type, or as a more complex type. Other than the EFI_IFR_TYPE_BOOLEAN and EFI_IFR_TYPE_NUM_x data types, all of the HII configuration data types are treated as a sequence of data.

**DataFilter.Numeric**

A sequence of data that must be interpreted as a one, two, four, or eight-byte wide numeric value. For instance, a definition of “Numeric:2” would indicate that the keyword is a two-byte numeric value. If no byte-size designation is specified, then the value may vary in size.

**DataFilter.String**

HII questions with EFI_IFR_TYPE_STRING type are treated as this type.

**DataFilter.Boolean**

HII questions with EFI_IFR_TYPE_BOOLEAN type are treated as this type.

**DataFilter.Date**

HII questions with EFI_IFR_TYPE_DATE type are treated as this type.

**DataFilter.Time**

HII questions with EFI_IFR_TYPE_TIME type are treated as this type.
The <UsageFilter> element defines the optional filter to use based on usage type when a request is made. If no filtering is desired, then this element must be omitted from the <KeywordRequest>.

**UsageType.ReadOnly**

The data for the keyword cannot be changed. It is intended solely for informational purposes, and can be used to read a setting that may be static or dynamic (e.g. CPU temperature).

**UsageType.ReadWrite**

The data for the keyword can be changed.

The <KeywordInfoRequest> element allows the callers to request some additional information of the keyword to be returned. <KeywordInfoRequest> element is used when user doesn’t know the information about the keyword and wants to get more information about this keyword. When <KeywordInfoRequest> element is specified with <KeywordRequest>, the <KeywordInfoResp> element will be specified with <KeywordResp> to return the info requested by <KeywordInfoRequest>.

The <KeywordInfoFilter> element is used to specify the additional information that caller wants to know about the keyword. Caller can specify any type of additional information he/she wants to know. When ‘All’ is specified, means all the supported information need to be returned.

The <DataType> element specifies the data type of a keyword, can refer to <DataFilter> for the detailed info the data types.

The <ValueAttribute> element specifies the value attribute of a keyword. Such as the value range of a keyword or the selectable values for a keyword.

<ValueRange> element specifies the variation range of a keyword value. Such as it can be specified for a keyword used in EFI_IFR_NUMERIC_OP, EFI_IFR_STRING_OP opcode. For EFI_IFR_NUMERIC_OP opcode, it specifies the maximum value, minimum value and increment or decrement step. For EFI_IFR_STRING_OP opcode, it specifies the maximum length and minimum length of the string can be input.

<ValueOption> element specifies all the (selectable) values and related string representation of these values for a keyword. Such as it can be specified for keyword used in EFI_IFR_ONE_OF_OP, EFI_IFR_ORDERED_LIST_OP opcode. For EFI_IFR_ONE_OF_OP, it specifies the all selectable values and the string representation of the values. The keyword value can be one of them.

For EFI_IFR_ORDERED_LIST_OP, it specifies all values and the string representation of the values. The keyword value can be the permutation and combination of these values. And for EFI_IFR_ORDERED_LIST_OP, its data type is Buffer, so can return the value type in a <ValueOption> to indicate the data stored in the Buffer is numeric as a one, two, four, or eight-byte wide.

The <Default> element specifies the default value of a keyword. Only the three standard defaults stores are supported including the standard defaults, the manufacturing defaults and the Safe defaults. If the keyword doesn’t have any type of defaults, then there is no default info returned. And if the keyword only has the standard default, then only the standard default information will be returned.

The <Display> element specifies the displayed prompt string of this keyword in the UI page.
35.2.1.6.1 An example of some basic keyword-related strings:

<KeywordRequest> to retrieve the current BIOS Vendor name:

   KEYWORD=BIOSVendor

35.2.1.6.2 A possible response might look like:

   x-UEFI-ns&KEYWORD=BIOSVendor&VALUE=AcmeBIOSCompany

If a request was made to retrieve all of the settings for a platform, a user would initiate a call to KeywordHandler->GetData() with the KeywordString and NamespaceId being NULL.

35.2.1.6.3 A possible response might look like:

   x-UEFI-ns&KEYWORD=BIOSVendor&VALUE=AcmeBIOSCompany&x-UEFI-extension-ACME&KEYWORD=SpecialSettingX&VALUE=3

In this case, the string returned tells us that there was one discovered keyword called “BIOSVendor” under the standard UEFI namespace and its value was “AcmeBIOS”. There was also an ACME branded namespace element which was discovered that had a keyword called “SpecialSettingX” whose value was 3.

35.2.1.6.4 An example to get more information of a keyword:

   KEYWORD=BIOSVendor&KEYWORDINFO=All

A possible response might look like:

   x-UEFI-ns&KEYWORD=BIOSVendor&VALUE=AcmeBIOSCompany&DataType=String&MAX=30&MIN=6&STANDARDDEFAULT=AcmeBIOSCompany&MFGDEFAULT=AcmeBIOSCompany

35.2.2 String Types

There are six string types. As can be seen from the BNF, the syntax of all is quite similar. The first three are used in communications between drivers and HII. The last three are used for analogous communication between external applications and HII.

<ConfigRequest>: This string is used by HII to request the current and any alternative configurations from a driver. It consists of routing information and only ampersand separated names.

<ConfigAltResp>: A string in this format is returned by the driver in response to a request to fill in a <ConfigRequest> string. The string consists of the current configuration followed by possibly several alternative configurations. The alternative configurations have the ALTCFG name / value pair in addition to the usual GUID, NAME, and PATH entries in the routing prefix. The ALTCFG value is a Default ID which is used to describe the alternative default configuration.

<ConfigResp>: A string in this format is handed by the HII to the driver to cause the driver to change its configuration. It consists of routing information and name / value pairs which correspond to the questions in the driver’s IFR. Only <ConfigResp> strings which refer to a driver in question may be handed to that driver. The driver shall reject all others.
<MultiConfigRequest>: A string in this format is handed to HII by an external application in order to request the current an alternate configurations of the system’s drivers. The format of this string is a series of <ConfigRequest> strings separated by ampersands. The HII’s job is to separate the requests and hand them off to the appropriate drivers (as indicated by the routing headers).

<MultiConfigAltResp>: A string in this format is handed back to an external application which has requested the current and alternate configurations of the system’s drivers. The format of this string is a series of <ConfigAltResp> strings separated by ampersands. The HII creates this string by concatenating the current and alternate configuration strings provided by each driver.

<MultiConfigResp>: A string in this format is handed to the HII in order to update the system’s configuration. Analogous to the other “Multi” string formats, its syntax is a series of ampersand separated <ConfigResp> strings. Upon receipt, the HII routes the <ConfigResp> strings to the corresponding drivers.

35.3 EFI Configuration Keyword Handler Protocol

This section provides a detailed description of the EFI Configuration Keyword Handler Protocol.

EFI_CONFIG_KEYWORD_HANDLER_PROTOCOL

Summary

The EFI_CONFIG_KEYWORD_HANDLER_PROTOCOL provides the mechanism to set and get the values associated with a keyword exposed through a x-UEFI- prefixed configuration language namespace.

GUID

```
#define EFI_CONFIG_KEYWORD_HANDLER_PROTOCOL_GUID \
{ 0xa8bad5, 0x3b8, 0xd19, \ 
{0xb1, 0x28, 0x7b, 0x8f, 0xe0, 0xda, 0xa5, 0x96 }}
```

Protocol Interface Structure

```
typedef struct _EFI_CONFIG_KEYWORD_HANDLER_PROTOCOL {
    EFI_CONFIG_KEYWORD_HANDLER_SET_DATA  SetData;
    EFI_CONFIG_KEYWORD_HANDLER_GET_DATA   GetData;
} EFI_CONFIG_KEYWORD_HANDLER_PROTOCOL;
```

Parameters

- **SetData**: Set the data associated with a particular configuration namespace keyword.
- **GetData**: Get the data associated with a particular configuration namespace keyword.

Description

The EFI_CONFIG_KEYWORD_HANDLER_PROTOCOL allows other components in the platform (e.g. Browser, Manageability Software, etc.) to retrieve and set configuration settings within the system.
Keywords are text elements which are associated with a particular configuration option within the platform. These keywords are intended to add semantic meaning to the configuration option they are attached to. The text associated for the keyword would be encoded in a UEFI configuration language. These languages are like French or German or Japanese, but are not designed for display purposes for an end-user. Instead each language serves as a namespace for the purposes of grouping and manipulating groups of platform configurations options. See Section 33.2.11.2 (Working with a UEFI Configuration Language) for more information.

Note: Not all configuration options will be associated with a keyword. Associating a keyword with a configuration option is at the discretion of the platform and/or the hardware vendor. For more information about keyword definitions associated with a UEFI namespace, see the UEFI Keyword Namespace Registry link in the UEFI Link Document.

**EFI_KEYвоORD_HANDLER _PROTOCOL.SetData()**

**Summary**
Set the data associated with a particular configuration namespace keyword.

**Prototype**
```
typedef
  EFI_STATUS
  (EFIAPI *EFI_KEYвоORD_HANDLER _SET_DATA) (    
    IN EFI_KEYвоORD_HANDLER_PROTOCOL *This,
    IN CONST EFI_STRING KeywordString,
    OUT EFI_STRING *Progress,
    OUT UINT32 *ProgressErr
  );
```

**Parameters**
- **This**
  Pointer to the EFI_KEYвоORD_HANDLER _PROTOCOL instance.
- **KeywordString**
  A null-terminated string in `<MultiKeywordResp>` format.
- **Progress**
  On return, points to a character in the KeywordString. Points to the string’s NULL terminator if the request was successful. Points to the most recent ‘&’ before the first failing name / value pair (or the beginning of the string if the failure is in the first name / value pair) if the request was not successful.
- **ProgressErr**
  If during the processing of the KeywordString there was a failure, this parameter gives additional information about the possible source of the problem. The various errors are defined in “Related Definitions” below.

**Description**
This function accepts a `<MultiKeywordResp>` formatted string, finds the associated keyword owners, creates a `<MultiConfigResp>` string from it and forwards it to the EFI_HII_ROUTING_PROTOCOL.RouteConfig function.
If there is an issue in resolving the contents of the `KeywordString`, then the function returns an error and also sets the `Progress` and `ProgressErr` with the appropriate information about where the issue occurred and additional data about the nature of the issue.

In the case when `KeywordString` containing multiple keywords, when an `EFI_NOT_FOUND` error is generated during processing the second or later keyword element, the system storage associated with earlier keywords is not modified. All elements of the `KeywordString` must successfully pass all tests for format and access prior to making any modifications to storage.

In the case when `EFI DEVICE_ERROR` is returned from the processing of a `KeywordString` containing multiple keywords, the state of storage associated with earlier keywords is undefined.

Related Definitions

```
//***********************************************************
// Progress Errors
//***********************************************************
#define KEYWORD_HANDLER_NO_ERROR          0x00000000
#define KEYWORD_HANDLER_NAMESPACE_ID_NOT_FOUND   0x00000001
#define KEYWORD_HANDLER_MALFORMED_STRING      0x00000002
#define KEYWORD_HANDLER_KEYWORD_NOT_FOUND      0x00000004
#define KEYWORD_HANDLER_INCOMPATIBLE_VALUE_DETECTED 0x00000008
#define KEYWORD_HANDLER_ACCESS_NOT_PERMITTED    0x00000010
#define KEYWORD_HANDLER_UNDEFINED_PROCESSING_ERROR 0x80000000
```

The `KEYWORD_HANDLER_x` values describe the error values returned in the `ProgressErr` field.

If no errors were encountered, then `KEYWORD_HANDLER_NO_ERROR` is returned with no bits are set.

If the `<NameSpaceId>` specified by the `KeywordString` was not found in any of the registered configuration data, the `KEYWORD_HANDLER_NAMESPACE_ID_NOT_FOUND` bit is set.

If there was an error in the parsing of the `KeywordString`, the `KEYWORD_HANDLER_MALFORMED_STRING` bit is set.

If there was a keyword specified in the `KeywordString` which was not found in any of the registered configuration data, `KEYWORD_HANDLER_KEYWORD_NOT_FOUND` bit is set.

If the value either passed into `KeywordString` (during a SetData operation) or the value discovered for the Keyword (during a GetData operation) did not match what was known to be valid for the defined keyword, the `KEYWORD_HANDLER_INCOMPATIBLE_VALUE_DETECTED` bit is set.

If there was an error as a result of a violation of system policy. For example trying to write a read-only element, the `KEYWORD_HANDLER_ACCESS_NOT_PERMITTED` bit is set.

If there was an undefined type of error in processing the passed in data, the `KEYWORD_HANDLER_UNDEFINED_PROCESSING_ERROR` bit is set.
### Status Codes Returned

<table>
<thead>
<tr>
<th>Code</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>EFI_SUCCESS</td>
<td>The specified action was completed successfully.</td>
</tr>
<tr>
<td>EFI_INVALID_PARAMETER</td>
<td>One or more of the following are <strong>TRUE</strong>: KeywordString is NULL. Parsing of the KeywordString resulted in an error. See Progress and ProgressErr for more data.</td>
</tr>
<tr>
<td>EFI_NOT_FOUND</td>
<td>An element of the KeywordString was not found. See ProgressErr for more data.</td>
</tr>
<tr>
<td>EFI_OUT_OF_RESOURCES</td>
<td>Required system resources could not be allocated. See ProgressErr for more data.</td>
</tr>
<tr>
<td>EFI_ACCESS_DENIED</td>
<td>The action violated system policy. See ProgressErr for more data.</td>
</tr>
<tr>
<td>EFI_DEVICE_ERROR</td>
<td>An unexpected system error occurred. See ProgressErr for more data.</td>
</tr>
</tbody>
</table>

### EFI_KEYWONRD_HANDLER _PROTOCOL.GetData()

**Summary**

Get the data associated with a particular configuration namespace keyword.

**Prototype**

```c
typedef
EFI_STATUS
(EFIAPI *EFI_KEYWORD_HANDLER _GET_DATA) (  
  IN EFI_KEYWORD_HANDLER_PROTOCOL *This,
  IN CONST EFI_STRING NameSpaceId, OPTIONAL
  IN CONST EFI_STRING KeywordString, OPTIONAL
  OUT EFI_STRING *Progress,
  OUT UINT32 *ProgressErr,
  OUT EFI_STRING *Results
);
```

**Parameters**

- **This**  
  Pointer to the **EFI_KEYWONRD_HANDLER _PROTOCOL** instance.

- **NamespaceId**  
  A null-terminated string containing the platform configuration language to search through in the system. If a NULL is passed in, then it is assumed that any platform configuration language with the prefix of “x-UEFI-” are searched.

- **KeywordString**  
  A null-terminated string in `<MultiKeywordRequest>` format. If a NULL is passed in the KeywordString field, all of the known keywords in the system for the NameSpaceId specified are returned in the Results field.

- **Progress**  
  On return, points to a character in the KeywordString. Points to the string’s NULL terminator if the request was successful. Points to the most recent ‘&’ before the first failing name /
value pair (or the beginning of the string if the failure is in the first name / value pair) if the request was not successful.

**ProgressErr**

If during the processing of the **KeywordString** there was a failure, this parameter gives additional information about the possible source of the problem. See the definitions in **SetData()** for valid value definitions.

**Results**

A null-terminated string in **<MultiKeywordResp>** format is returned which has all the values filled in for the keywords in the **KeywordString**. This is a callee-allocated field, and must be freed by the caller after being used.

**Description**

This function accepts a **<MultiKeywordRequest>** formatted string, finds the underlying keyword owners, creates a **<MultiConfigRequest>** string from it and forwards it to the **EFI_HII_ROUTING_PROTOCOL.ExtractConfig** function.

If there is an issue in resolving the contents of the **KeywordString**, then the function returns an **EFI_INVALID_PARAMETER** and also set the **Progress** and **ProgressErr** with the appropriate information about where the issue occurred and additional data about the nature of the issue.

In the case when **KeywordString** is NULL, or contains multiple keywords, or when **EFI_NOT_FOUND** is generated while processing the keyword elements, the **Results** string contains values returned for all keywords processed prior to the keyword generating the error but no values for the keyword with error or any following keywords.

**Status Codes Returned**

<table>
<thead>
<tr>
<th>Status Code</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>EFI_SUCCESS</strong></td>
<td>The specified action was completed successfully.</td>
</tr>
<tr>
<td><strong>EFI_INVALID_PARAMETER</strong></td>
<td>One or more of the following are <strong>TRUE</strong>: <strong>Progress</strong>, <strong>ProgressErr</strong>, or <strong>Results</strong> is <strong>NULL</strong>. Parsing of the <strong>KeywordString</strong> resulted in an error. See <strong>Progress</strong> and <strong>ProgressErr</strong> for more data.</td>
</tr>
<tr>
<td><strong>EFI_NOT_FOUND</strong></td>
<td>An element of the <strong>KeywordString</strong> was not found. See <strong>ProgressErr</strong> for more data.</td>
</tr>
<tr>
<td><strong>EFI_NOT_FOUND</strong></td>
<td>The <strong>NamespaceId</strong> specified was not found. See <strong>ProgressErr</strong> for more data.</td>
</tr>
<tr>
<td><strong>EFI_OUT_OF_RESOURCES</strong></td>
<td>Required system resources could not be allocated. See <strong>ProgressErr</strong> for more data.</td>
</tr>
<tr>
<td><strong>EFI_ACCESS_DENIED</strong></td>
<td>The action violated system policy. See <strong>ProgressErr</strong> for more data.</td>
</tr>
<tr>
<td><strong>EFI_DEVICE_ERROR</strong></td>
<td>An unexpected system error occurred. See <strong>ProgressErr</strong> for more data.</td>
</tr>
</tbody>
</table>
35.4 EFI HII Configuration Routing Protocol

EFI_HII_CONFIG_ROUTING_PROTOCOL

Summary
The EFI HII Configuration Routing Protocol manages the movement of configuration data from drivers to configuration applications. It then serves as the single point to receive configuration information from configuration applications, routing the results to the appropriate drivers.

GUID
#define EFI_HII_CONFIG_ROUTING_PROTOCOL_GUID  
   { 0x587e72d7, 0xcc50, 0x4f79,  
   { 0x82, 0x09, 0xca, 0x29, 0x1f, 0xc1, 0xa1, 0x0f } }

Protocol Interface Structure
typedef struct {
   EFI_HII_EXTRACT_CONFIG     ExtractConfig;
   EFI_HII_EXPORT_CONFIG      ExportConfig
   EFI_HII_ROUTE_CONFIG       RouteConfig;
   EFI_HII_BLOCK_TO_CONFIG    BlockToConfig;
   EFI_HII_CONFIG_TO_BLOCK    ConfigToBlock;
   EFI_HII_GET_ALT_CFG        GetAltConfig;
} EFI_HII_CONFIG_ROUTING_PROTOCOL;

Related Definitions
None

Parameters

Description
This protocol defines the configuration routing interfaces between external applications and the HII. There may only be one instance of this protocol in the system.

EFI_HII_CONFIG_ROUTING_PROTOCOL.ExtractConfig()

Summary
This function allows a caller to extract the current configuration for one or more named elements from one or more drivers.
Prototype

```c
typedef EFI_STATUS (EFIAPICALLTYPE EFI_HII_EXTRACT_CONFIG ) (  
    IN CONST EFI_HII_CONFIG_ROUTING_PROTOCOL *This,  
    IN CONST EFI_STRING Request,  
    OUT EFI_STRING *Progress,  
    OUT EFI_STRING *Results  
);  
```

Parameters

This

Points to the `EFI_HII_CONFIG_ROUTING_PROTOCOL` instance.

Request

A null-terminated string in `<MultiConfigRequest>` format.

Progress

On return, points to a character in the Request string. Points to the string’s null terminator if request was successful. Points to the most recent ‘&’ before the first failing name / value pair (or the beginning of the string if the failure is in the first name / value pair) if the request was not successful.

Results

A null-terminated string in `<MultiConfigAltResp>` format which has all values filled in for the names in the Request string.

Description

This function allows the caller to request the current configuration for one or more named elements from one or more drivers. The resulting string is in the standard HII configuration string format. If Successful Results contains an equivalent string with “=“ and the values associated with all names added in.

The expected implementation is for each `<ConfigRequest>` substring in the Request, call the HII Configuration Access Protocol `ExtractConfig` function for the driver corresponding to the `<ConfigHdr>` at the start of the `<ConfigRequest>` substring. The request fails if no driver matches the `<ConfigRequest>` substring.

Note: Alternative configuration strings may also be appended to the end of the current configuration string. If they are, they must appear after the current configuration. They must contain the same routing (GUID, NAME, PATH) as the current configuration string. They must have an additional description indicating the type of alternative configuration the string represents, “ALT_CFG=<AltCfgId>“. The `<AltCfgId>` is a reference to a Default ID which stipulates the type of Default being referenced such as `EFI_HII_DEFAULT_CLASS_STANDARD`.

As an example, assume that the Request string is:

```
GUID=…&PATH=…&Fred&George&Ron&Neville
```

A result might be:
GUID=...&PATH=...&Fred=16&George=16&Ron=12&Neville=11
GUID=...&PATH=...&ALTCFG=0037&Fred=12&Neville=7

Note: For the output Results, the value filled in the names in the Request string with
<MultiConfigAltResp> format may change when called multiple times due to some data
being of a dynamic nature.

Status Codes Returned

EFI_SUCCESS The Results string is filled with the values corresponding to all requested names.
EFI_OUT_OF_RESOURCES Not enough memory to store the parts of the results that must be stored awaiting possible future protocols.
EFI_NOT_FOUND Routing data doesn’t match any known driver. Progress set to the “G” in “GUID” of the routing header that doesn’t match. Note: There is no requirement that all routing data be validated before any configuration extraction.
EFI_INVALID_PARAMETER Illegal syntax. Progress set to most recent “&” before the error or the beginning of the string.
EFI_INVALID_PARAMETER The ExtractConfig function of the underlying HII Configuration Access Protocol returned EFI_INVALID_PARAMETER. Progress set to most recent “&” before the error or the beginning of the string.
EFI_ACCESS_DENIED The action violated a system policy.

EFI_HII_CONFIG_ROUTING_PROTOCOL.ExportConfig()

Summary
This function allows the caller to request the current configuration for the entirety of the current HII database and returns the data in a null-terminated string.

Prototype

typedef

EFI_STATUS

(EIFIAPIC * EFI_HII_EXPORT_CONFIG ) (  
    IN CONST EFI_HII_CONFIG_ROUTING_PROTOCOL *This,
    OUT EFI_STRING *Results
);

Parameters

This

Points to the EFI_HII_CONFIG_ROUTING_PROTOCOL instance.

Results

A null-terminated string in <MultiConfigAltResp> format which has all values filled in for the entirety of the current HII database.
Description
This function allows the caller to request the current configuration for all of the current HII database. The results include both the current and alternate configurations as described in `ExtractConfig()` above.

`EFI_HII_CONFIG_ACCESS_PROTOCOL.ExtractConfig()` interfaces below.

Status Codes Returned

<table>
<thead>
<tr>
<th>Status Code</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>EFI_SUCCESS</td>
<td>The Results string is filled with the values corresponding to all requested names.</td>
</tr>
<tr>
<td>EFI_OUT_OF_RESOURCES</td>
<td>Not enough memory to store the parts of the results that must be stored awaiting possible future protocols.</td>
</tr>
<tr>
<td>EFI_INVALID_PARAMETER</td>
<td>For example, passing in a NULL for the Results parameter would result in this type of error.</td>
</tr>
</tbody>
</table>

`EFI_HII_CONFIG_ROUTING_PROTOCOL.RouteConfig()`

Summary
This function processes the results of processing forms and routes it to the appropriate handlers or storage.

Prototype
```c
typedef EFI_STATUS
(EFIAPI * EFI_HII_ROUTE_CONFIG ) ( 
    IN CONST EFI_HII_CONFIG_ROUTING_PROTOCOL *This,
    IN CONST EFI_STRING Configuration,
    OUT EFI_STRING *Progress
);
```

Parameters
- **This**
  Points to the `EFI_HII_CONFIG_ROUTING_PROTOCOL` instance.
- **Configuration**
  A null-terminated string in `<MultiConfigResp>` format.
- **Progress**
  A pointer to a string filled in with the offset of the most recent ‘&’ before the first failing name / value pair (or the beginning of the string if the failure is in the first name / value pair) or the terminating NULL if all was successful.

Description
This function routes the results of processing forms to the appropriate targets. It scans for `<ConfigHdr>` within the string and passes the header and subsequent body to the driver whose location is described in the `<ConfigHdr>`. Many `<ConfigHdr>`s may appear as a single request.
The expected implementation is to hand off the various `<ConfigResp>` substrings to the Configuration Access Protocol `RouteConfig` routine corresponding to the driver whose routing information is defined by the `<ConfigHdr>` in turn.

### Status Codes Returned

- **EFI_SUCCESS**: The results have been distributed or are awaiting distribution.
- **EFI_OUT_OF_RESOURCES**: Not enough memory to store the parts of the results that must be stored awaiting possible future protocols.
- **EFI_INVALID_PARAMETERS**: Passing in a NULL for the `Configuration` parameter would result in this type of error.
- **EFI_NOT_FOUND**: Target for the specified routing data was not found.
- **EFI_ACCESS_DENIED**: The action violated a system policy.

### EFI_HII_CONFIG_ROUTING_PROTOCOL.BlockToConfig()

#### Summary
This helper function is to be called by drivers to map configuration data stored in byte array (“block”) formats such as UEFI Variables into current configuration strings.

#### Prototype

```c
typedef EFI_STATUS
(EIFIAPI * EFI_HII_BLOCK_TO_CONFIG ) (  
    IN CONST EFI_HII_CONFIG_ROUTING_PROTOCOL *This,  
    IN CONST EFI_STRING ConfigRequest,  
    IN CONST UINT8 *Block,  
    IN CONST UINTN BlockSize,  
    OUT EFI_STRING *Config,  
    OUT EFI_STRING *Progress  
);  
```

#### Parameters

- **This**: Points to the `EFI_HII_CONFIG_ROUTING_PROTOCOL` instance.
- **ConfigRequest**: A null-terminated string in `<ConfigRequest>` format.
- **Block**: Array of bytes defining the block’s configuration.
- **BlockSize**: Length in bytes of `Block`. 
Config
Filled-in configuration string. String allocated by the function. Returned only if call is successful. The null-terminated string will be in `<ConfigResp>` format.

Progress
A pointer to a string filled in with the offset of the most recent ‘&’ before the first failing name / value pair (or the beginning of the string if the failure is in the first name / value pair) or the terminating NULL if all was successful.

Description
This function extracts the current configuration from a block of bytes. To do so, it requires that the ConfigRequest string consists of a list of `<BlockName>` formatted names. It uses the offset in the name to determine the index into the Block to start the extraction and the width of each name to determine the number of bytes to extract. These are mapped to a string using the equivalent of the C “%x” format (with optional leading spaces).

The call fails if, for any (offset, width) pair in ConfigRequest, offset+value >= BlockSize.

Status Codes Returned

EFI_SUCCESS The request succeeded. Progress points to the null terminator at the end of the ConfigRequest string.

EFI_OUT_OF_RESOURCES Not enough memory to allocate Config. Progress points to the first character of ConfigRequest.

EFI_INVALID_PARAMETERS Passing in a NULL for the ConfigRequest or Block parameter would result in this type of error. Progress points to the first character of ConfigRequest.

EFI_NOT_FOUND Target for the specified routing data was not found. Progress points to the “G” in "GUID" of the errant routing data.

EFI_DEVICE_ERROR Block not large enough. Progress undefined.

EFI_INVALID_PARAMETER Encountered non `<BlockName>` formatted string. Block is left updated and Progress points at the ‘&’ preceding the first non-<BlockName>.

EFI_HII_CONFIG_ROUTING_PROTOCOL.ConfigToBlock()

Summary
This helper function is to be called by drivers to map configuration strings to configurations stored in byte array ("block") formats such as UEFI Variables.
Prototype

typedef

EFI_STATUS

(EIFIAPI * EFI_HII_CONFIG_TO_BLOCK ) (  
IN   CONST EFI_HII_CONFIG_ROUTING_PROTOCOL *This,
IN   CONST EFI_STRING *ConfigResp,
IN OUT CONST UINT8 *Block,
IN OUT UINTN *BlockSize,
OUT  EFI_STRING *Progress
);

Parameters

This
Points to the EFI_HII_CONFIG_ROUTING_PROTOCOL instance.

ConfigResp
A null-terminated string in <ConfigResp> format.

Block
A possibly null array of bytes representing the current block. Only bytes referenced
in the ConfigResp string in the block are modified. If this parameter is null or if the
*BlockSize parameter is (on input) shorter than required by the Configuration
string, only the BlockSize parameter is updated and an appropriate status (see
below) is returned.

BlockSize
The length of the Block in units of UINT8. On input, this is the size of the Block. On
output, if successful, contains the largest index of the modified byte in the Block, or
the required buffer size if the Block is not large enough.

Progress
On return, points to an element of the ConfigResp string filled in with the offset of
the most recent ‘&’ before the first failing name / value pair (or the beginning of the
string if the failure is in the first name / value pair) or the terminating NULL if all was
successful.

Description
This function maps a configuration containing a series of <BlockConfig> formatted name value pairs in
ConfigResp into a Block so it may be stored in a linear mapped storage such as a UEFI Variable. If
present, the function skips GUID, NAME, and PATH in <ConfigResp>. It stops when it finds a non-
<BlockConfig> name / value pair (after skipping the routing header) or when it reaches the end of the
string.

Example
Assume an existing block containing:

00 01 02 03 04 05

And the ConfigResp string is:
The results are

55 AA 02 07 04 05

Status Codes Returned

- **EFI_SUCCESS**: The request succeeded. Progress points to the null terminator at the end of the `ConfigResp` string.
- **EFI_OUT_OF_RESOURCES**: Not enough memory to allocate `Config`. Progress points to the first character of `ConfigResp`.
- **EFI_INVALID_PARAMETER**: Passing in a NULL for the `ConfigResp` or `Block` parameter would result in this type of error. Progress points to the first character of `ConfigResp`.
- **EFI_NOT_FOUND**: Target for the specified routing data was not found. Progress points to the “G” in “GUID” of the errant routing data.
- **EFI_BUFFER_TOO_SMALL**: Block not large enough. Progress undefined. `BlockSize` is updated with the required buffer size.
- **EFI_INVALID_PARAMETER**: Encountered non `<BlockName>` formatted name / value pair. `Block` is left updated and Progress points at the `&` preceding the first non- `<BlockName>`.

**EFI_HII_CONFIG_ROUTING_PROTOCOL.GetAltCfg()**

**Summary**

This helper function is to be called by drivers to extract portions of a larger configuration string.

**Prototype**

```c
typedef EFI_STATUS (EFIAPI *EFI_HII_GET_ALT_CFG)(
  IN   CONST EFI_HII_CONFIG_ROUTING_PROTOCOL *This,
  IN   CONST EFI_STRING     ConfigResp,
  IN   CONST EFI_GUID       *Guid,
  IN   CONST EFI_STRING     Name,
  IN   CONST EFI_DEVICE_PATH_PROTOCOL *DevicePath,
  IN   CONST EFI_STRING     AltCfgId,
  OUT  EFI_STRING           *AltCfgResp
);
```

**Parameters**

- **This**
  
  Points to the `EFI_HII_CONFIG_ROUTING_PROTOCOL` instance.

- **ConfigResp**
  
  A null-terminated string in `<ConfigAltResp>` format.
Guid

A pointer to the GUID value to search for in the routing portion of the ConfigResp string when retrieving the requested data. If Guid is NULL, then all GUID values will be searched for.

Name

A pointer to the NAME value to search for in the routing portion of the ConfigResp string when retrieving the requested data. If Name is NULL, then all Name values will be searched for.

DevicePath

A pointer to the PATH value to search for in the routing portion of the ConfigResp string when retrieving the requested data. If DevicePath is NULL, then all DevicePath values will be searched for.

AltCfgId

A pointer to the ALTCFG value to search for in the routing portion of the ConfigResp string when retrieving the requested data. If this parameter is NULL, then the current setting will be retrieved.

AltCfgResp

A pointer to a buffer which will be allocated by the function which contains the retrieved string as requested. This buffer is only allocated if the call was successful. The null-terminated string will be in <ConfigResp> format.

Description

This function retrieves the requested portion of the configuration string from a larger configuration string. This function will use the Guid, Name, and DevicePath parameters to find the appropriate section of the ConfigResp string. Upon finding this portion of the string, it will use the AltCfgId parameter to find the appropriate instance of data in the ConfigResp string. Once found, the found data will be copied to a buffer which is allocated by the function so that it can be returned to the caller. The caller is responsible for freeing this allocated buffer.

Status Codes Returned

EFI_SUCCESS The request succeeded. The requested data was extracted and placed in the newly allocated AltCfgResp buffer.

EFI_OUT_OF_RESOURCES Not enough memory to allocate AltCfgResp.

EFI_INVALID_PARAMETER Passing in a NULL for the ConfigResp or AltCfgResp would result in this type of error.
35.5 EFI HII Configuration Access Protocol

**EFI_HII_CONFIG_ACCESS_PROTOCOL**

**Summary**

The EFI HII configuration routing protocol invokes this type of protocol when it needs to forward requests to a driver’s configuration handler. This protocol is published by drivers providing and receiving configuration data from HII. The `ExtractConfig()` and `RouteConfig()` functions are typically invoked by the driver which implements the HII Configuration Routing Protocol. The `Callback()` function is typically invoked by the Forms Browser.

If the protocol functions modify active form set, they must not change layout and size of the existing variable stores. The forms browser processes updated IFR package in accordance with the following rules:

1. If active form set no longer exists, the behavior is browser specific. The browser identifies form set using a combination of the form set GUID and device path associated with the package list containing the form set.
2. If form set update has been initiated by the `Callback()` function, the browser executes action requested by the function. See `EFI_HII_CONFIG_ACCESS_PROTOCOL.CallBack()` section for additional details regarding browser action requests.

**Note:** *If browser action implies saving of the modified questions values, the browser will use uncommitted data associated with the old form set instance. The HII Configuration Access implementation is responsible for properly handling such requests.*

3. The browser performs standard processing steps that are performed on a form set prior to displaying it (including reading question values and generating `EFI_BROWSER_ACTION_FORM_OPEN` and `EFI_BROWSER_ACTION_FORM_RETRIEVE` callbacks).
4. If there is an uncommitted browser data associated with an active form set, the browser applies it, matching variable stores by their identifiers. If variable store no longer exists, the uncommitted data for this store is discarded.

**Note:** *Changing layout or size of the existing variable stores during form set update is not allowed and can lead to unpredictable results.*

5. The browser applies prior browsing history, matching forms by their identifiers. If a form saved in the browsing history no longer exists, the behavior is browser-specific.
6. If all forms in the browsing history have been matched, the browser sets selection on a question that was active prior to the form set update, matching question by its identifier. If question does not exist, the first question on the form is selected.
GUID

#define EFI_HII_CONFIG_ACCESS_PROTOCOL_GUID
{ 0x330d4706, 0xf2a0, 0x4e4f, \
 {0xa3,0x69, 0xb6, 0x6f,0xa8, 0xd5, 0x43, 0x85}}

Protocol Interface Structure

typedef struct {
 EFI_HII_ACCESS_EXTRACT_CONFIG ExtractConfig;
 EFI_HII_ACCESS_ROUTE_CONFIG RouteConfig;
 EFI_HII_ACCESS_FORM_CALLBACK Callback;
 } EFI_HII_CONFIG_ACCESS_PROTOCOL;

Related Definitions
None

Parameters

ExtractConfig
This function breaks apart the request strings routing them to the appropriate
drivers. This function is analogous to the similarly named function in the HII Routing
Protocol.

RouteConfig
This function breaks apart the results strings and returns configuration information
as specified by the request.

Callback
This function is called from the configuration browser to communicate certain
activities that were initiated by a user.

Description
This protocol provides a callable interface between the HII and drivers. Only drivers which provide IFR
data to HII are required to publish this protocol.

EFI_HII_CONFIG_ACCESS_PROTOCOL.ExtractConfig()

Summary
This function allows a caller to extract the current configuration for one or more named elements from
the target driver.
Prototype

typedef
EFI_STATUS
(EFIAPI * EFI_HII_ACCESS_EXTRACT_CONFIG ) ( 
IN CONST EFI_HII_CONFIG_ACCESS_PROTOCOL   *This,
IN CONST EFI_STRING Request,
OUT EFI_STRING *Progress,
OUT EFI_STRING *Results
);

Parameters

This

Points to the EFI_HII_CONFIG_ACCESS_PROTOCOL.

Request

A null-terminated string in ConfigRequest format. Note that this includes the routing information as well as the configurable name / value pairs. It is invalid for this string to be in MultiConfigRequest format.

If a NULL is passed in for the Request field, all of the settings being abstracted by this function will be returned in the Results field. In addition, if a ConfigHdr is passed in with no request elements, all of the settings being abstracted for that particular ConfigHdr reference will be returned in the Results Field.

Progress

On return, points to a character in the Request string. Points to the string’s null terminator if request was successful. Points to the most recent ‘&’ before the first failing name / value pair (or the beginning of the string if the failure is in the first name / value pair) if the request was not successful.

Results

A null-terminated string in MultiConfigAltResp format which has all values filled in for the names in the Request string. String to be allocated by the called function.

Description

This function allows the caller to request the current configuration for one or more named elements. The resulting string is in ConfigResp format.

In order to support forms processors other than a Forms Browser, the configuration returned by this function must not depend on context in which the function is used. In particular, it must not depend on the current state of the Forms Browser (including any uncommitted state information) and actions performed by the driver callbacks invoked prior to the ExtractConfig call. Section 33.2.1.8 provides additional details regarding forms browser/processor.

Any and all alternative configuration strings shall also be appended to the end of the current configuration string. If they are, they must appear after the current configuration. They must contain the same routing (GUID, NAME, PATH) as the current configuration string. They must have an additional description indicating the type of alternative configuration the string represents,
"ALTCFG=<AltCfgId>". The <AltCfgId> is a reference to a Default ID which stipulates the type of Default being referenced such as EFI_HII_DEFAULT_CLASS_STANDARD.

As an example, assume that the Request string is:

```
GUID=...&PATH=...&Fred&George&Ron&Neville
```

A result might be:

```
GUID=...&PATH=...&Fred=16&George=16&Ron=12&Neville=11&GUID=...&PATH=...&ALTCFG=0037&Fred=12&Neville=7
```

This function allows the caller to request the current configuration for one or more named elements. The resulting string is in <ConfigAltResp> format.

Any and all alternative configuration strings shall also be appended to the end of the current configuration string. If they are, they must appear after the current configuration. They must contain the same routing (GUID, NAME, PATH) as the current configuration string. They must have an additional description indicating the type of alternative configuration the string represents, “ALTCFG=<AltCfgId>”. The <AltCfgId> is a reference to a Default ID which stipulates the type of Default being referenced such as EFI_HII_DEFAULT_CLASS_STANDARD.

As an example, assume that the Request string is:

```
GUID=...&PATH=...&Fred&George&Ron&Neville
```

A result might be:

```
GUID=...&PATH=...&Fred=16&George=16&Ron=12&Neville=11&GUID=...&PATH=...&ALTCFG=0037&Fred=12&Neville=7
```

**Note:** For the output Results, the value filled in the names in the Request string with <ConfigAltResp> format may change when called multiple times due to some data being of a dynamic nature.

**Status Codes Returned**

- **EFI_SUCCESS**
  
  The Results string is filled with the values corresponding to all requested names.

- **EFI_OUT_OF_RESOURCES**
  
  Not enough memory to store the parts of the results that must be stored awaiting possible future protocols.

- **EFI_NOT_FOUND**
  
  A configuration element matching the routing data is not found. Progress set to the first character in the routing header.

- **EFI_INVALID_PARAMETER**
  
  Illegal syntax. Progress set to most recent "&" before the error or the beginning of the string.

- **EFI_INVALID_PARAMETER**
  
  Unknown name. Progress points to the "&" before the name in question.

- **EFI_INVALID_PARAMETER**
  
  If Results or Progress is NULL.

- **EFI_ACCESS_DENIED**
  
  The action violated a system policy.

- **EFI_DEVICE_ERROR**
  
  Failed to extract the current configuration for one or more named elements.
EFI_HII_CONFIG_ACCESS_PROTOCOL.RouteConfig()

Summary
This function processes the results of changes in configuration for the driver that published this protocol.

Prototype

```c
typedef
    EFI_STATUS
    (EFIAPI * EFI_HII_ACCESS_ROUTE_CONFIG ) (  
        IN CONST EFI_HII_CONFIG_ACCESS_PROTOCOL  *This,
        IN CONST EFI_STRING  Configuration,
        OUT EFI_STRING  *Progress
    );
```

Parameters

- **This**
  Points to the `EFI_HII_CONFIG_ACCESS_PROTOCOL`

- **Configuration**
  A null-terminated string in `<ConfigResp>` format.

- **Progress**
  a pointer to a string filled in with the offset of the most recent ‘&’ before the first failing name / value pair (or the beginning of the string if the failure is in the first name / value pair) or the terminating NULL if all was successful.

Description
This function applies changes in a driver's configuration. Input is a **Configuration**, which has the routing data for this driver followed by name / value configuration pairs. The driver must apply those pairs to its configurable storage.

In order to support forms processors other than a Forms Browser, the way in which configuration data is applied must not depend on context in which the function is used. In particular, it must not depend on the current state of the Forms Browser (including any uncommitted state information) and actions performed by the driver callbacks invoked prior to the **RouteConfig** call. Section 33.2.1.8 provides additional details regarding forms browser/processor.

If the driver's configuration is stored in a linear block of data and the driver's name / value pairs are in `<BlockConfig>` format, it may use the `ConfigToBlock` helper function (above) to simplify the job.
Status Codes Returned

<table>
<thead>
<tr>
<th>Code</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>EFI_SUCCESS</td>
<td>The results have been distributed or are awaiting distribution.</td>
</tr>
<tr>
<td>EFI_OUT_OF_RESOURCES</td>
<td>Not enough memory to store the parts of the results that must be stored awaiting possible future protocols.</td>
</tr>
<tr>
<td>EFI_INVALID_PARAMETER</td>
<td>If Configuration or Progress is NULL.</td>
</tr>
<tr>
<td>EFI_NOT_FOUND</td>
<td>Target for the specified routing data was not found.</td>
</tr>
<tr>
<td>EFI_ACCESS_DENIED</td>
<td>The action violated a system policy.</td>
</tr>
</tbody>
</table>

EFI_HII_CONFIG_ACCESS_PROTOCOL.CallBack()

Summary
This function is called to provide results data to the driver.

Prototype
typedef
    EFI_STATUS
(EFI_API *EFI_HII_ACCESS_FORM_CALLBACK) (  
    IN   CONST EFI_HII_CONFIG_ACCESS_PROTOCOL *This,
    IN   EFI_BROWSER_ACTION Action,
    IN   EFI_QUESTION_ID QuestionId,
    IN   UINT8 Type
    IN OUT EFI_IFR_TYPE_VALUE *Value,
    OUT   EFI_BROWSER_ACTION_REQUEST *ActionRequest,
);

Parameters

This
Points to the EFI_HII_CONFIG_ACCESS_PROTOCOL.

Action
Specifies the type of action taken by the browser. See EFI_BROWSER_ACTION_x in “Related Definitions” below.

QuestionId
A unique value which is sent to the original exporting driver so that it can identify the type of data to expect. The format of the data tends to vary based on the opcode that generated the callback.

Type
The type of value for the question. See EFI_IFR_TYPE_x in EFI_IFR_ONE_OF_OPTION.

Value
A pointer to the data being sent to the original exporting driver. The type is specified by Type. Type EFI_IFR_TYPE_VALUE is defined in EFI_IFR_ONE_OF_OPTION.
**ActionRequest**

On return, points to the action requested by the callback function. Type `EFI_BROWSER_ACTION_REQUEST` is specified in `SendForm()` in the Form Browser Protocol.

**Description**

This function is called by the forms browser in response to a user action on a question which has the `EFI_IFR_FLAG_CALLBACK` bit set in the `EFI_IFR_QUESTION_HEADER`. The user action is specified by `Action`. Depending on the action, the browser may also pass the question value using `Type` and `Value`. Upon return, the callback function may specify the desired browser action.

The browser maintains uncommitted browser data (modified and unsaved question values) across Callback function boundaries. Callback function may change unsaved question values using one of the following methods:

- Current question's value may be changed by updating the `Value` parameter.
- Values of other questions from the active formset can be changed using `EFI_FORM_BROWSER2_PROTOCOL.BrowserCallback()` interface.

**Note:** Modification of the question values by the Callback function without notifying the browser using one of the above mentioned methods can lead to unpredictable browser behavior.

Callback function may request configuration update from the browser by returning an appropriate `ActionRequest`.

In order to save uncommitted data, driver should return one of the `_SUBMIT` actions or `_APPLY` action. The browser will then write all modified question values (in case of the `_SUBMIT` actions) or modified question values from an active form (in case of the `_APPLY` action) to storage using `RouteConfig()` function. This will include questions modified prior to an invocation of the `Callback()` function as well as questions modified by the `Callback()` function.

The behavior of the `ExtractConfig` and `RouteConfig` functions must not depend on the actions performed by this function.

Callback functions should return `EFI_UNSUPPORTED` for all values of `Action` that they do not support.
Related Definitions

typedef UINTN EFI_BROWSER_ACTION;

#define EFI_BROWSER_ACTION_CHANGING       0
#define EFI_BROWSER_ACTION_CHANGED        1
#define EFI_BROWSER_ACTION_RETRIEVE       2
#define EFI_BROWSER_ACTION_FORM_OPEN       3
#define EFI_BROWSER_ACTION_FORM_CLOSE      4
#define EFI_BROWSER_ACTION_SUBMITTED       5
#define EFI_BROWSER_ACTION_DEFAULT_STANDARD   0x1000
#define EFI_BROWSER_ACTION_DEFAULT_MANUFACTURING 0x1001
#define EFI_BROWSER_ACTION_DEFAULT_SAFE     0x1002
#define EFI_BROWSER_ACTION_DEFAULT_PLATFORM   0x2000
#define EFI_BROWSER_ACTION_DEFAULT_HARDWARE   0x3000
#define EFI_BROWSER_ACTION_DEFAULT_FIRMWARE   0x4000

The following table describes the behavior of the callback for each question type.

**Table 35-1 Callback Behavior**

<table>
<thead>
<tr>
<th>Question Type</th>
<th>Type</th>
<th>Action</th>
</tr>
</thead>
<tbody>
<tr>
<td>Action Button</td>
<td>EFI_IFR_TYPE_ACTION</td>
<td>No special behavior. If the short form of the opcode is used, then the value will be a string identifier of zero.</td>
</tr>
<tr>
<td>Checkbox</td>
<td>EFI_IFR_TYPE_BOOLEAN</td>
<td>No special behavior</td>
</tr>
<tr>
<td>Cross-Reference</td>
<td>EFI_IFR_TYPE_REF, EFI_IFR_TYPE_UNDEFINED</td>
<td>CHANGING: If EFI_UNSUPPORTED or EFI_SUCCESS, the updated cross-reference is taken. Any other error the cross-reference will not be taken. CHANGED: Never called. RETRIEVE: Called before displaying the cross-reference. Error codes ignored. The Ref field of the Value parameter is initialized with the REF question’s value prior to CHANGING and RETRIEVE.</td>
</tr>
<tr>
<td>Date</td>
<td>EFI_IFR_TYPE_DATE</td>
<td>No special behavior</td>
</tr>
<tr>
<td>Numeric, One-Of</td>
<td>EFI_IFR_TYPE_NUM_SIZE_8, EFI_IFR_TYPE_NUM_SIZE_16, EFI_IFR_TYPE_NUM_SIZE_32, EFI_IFR_TYPE_NUM_SIZE_64</td>
<td>No special behavior</td>
</tr>
<tr>
<td>Ordered-List</td>
<td>EFI_IFR_TYPE_BUFFER</td>
<td>No special behavior</td>
</tr>
<tr>
<td>String, Password</td>
<td>EFI_IFR_TYPE_STRING</td>
<td>No special behavior</td>
</tr>
<tr>
<td>Time</td>
<td>EFI_IFR_TYPE_DATE</td>
<td>No special behavior</td>
</tr>
</tbody>
</table>
The value **EFI_BROWSER_ACTION_CHANGING** is called before the browser changes the value in the display (for questions which have a value) or takes an action (in the case of an action button or cross-reference). If the callback returns **EFI_UNSUPPORTED**, then the browser will use the value passed to `Callback()` and ignore the value returned by `Callback()`. If the callback returns **EFI_SUCCESS**, then the browser will use the value returned by `Callback()`. If any other error is returned, then the browser will not update the current question value. `ActionRequest` is used. The `Value` represents the updated value. The changes here should not be finalized until the user submits the results.

The value **EFI_BROWSER_ACTION_CHANGED** is called after the browser has changed its internal copy of the question value and displayed it (if appropriate). For action buttons, this is called after the value has been processed. For cross-references, this is never called. Errors returned are ignored. `ActionRequest` is used. The changes here should not be finalized until the user submits the results.

The value **EFI_BROWSER_ACTION_RETRIEVE** is called after the browser has read the current question value, but before it has been displayed. If the callback returns **EFI_UNSUPPORTED** or any other error then the original value is used. If **EFI_SUCCESS** is returned, then the updated value is used.

The value **EFI_BROWSER_ACTION_FORM_OPEN** is called for each question on a form prior to its value being retrieved or displayed. If a question appears on more than one form, and the Forms Browser supports more than one form being active simultaneously, this may be called more than once, even prior to any **EFI_BROWSER_ACTION_FORM_CLOSE** callback.

**NOTE:** **EFI_FORM_BROWSER2_PROTOCOL::BrowserCallback()** cannot be used with this browser action because question values have not been retrieved yet.

The value **EFI_BROWSER_ACTION_FORM_CLOSE** is called for each question on a form after the processing of any submit actions for that form. If a question appears on more than one form, and the Forms Processor supports more than one form being active simultaneously, this will be called more than once.

The value **EFI_BROWSER_ACTION_SUBMITTED** is called after Browser submits the modified question value. `ActionRequest` is ignored.

When `Action` specifies one of the "default" actions, such as **EFI_BROWSER_ACTION_DEFAULT_STANDARD**, etc. it indicates that the Forms Processor is attempting to retrieve the default value for the specified question. The proposed default value is passed in using `Type` and `Value` and reflects the value which the Forms Processor was able to select based on the lower-priority defaulting methods (see Section 33.2.5.8). If the function returns **EFI_SUCCESS**, then the updated value will be used. If the function does not have an updated default value for the specified question or specified default store, or does not provide any support for the actions, it should return **EFI_UNSUPPORTED**, and the returned value will be ignored.

The **DEFAULT_PLATFORM**, **DEFAULT_HARDWARE** and **DEFAULT_FIRMWARE** represent ranges of 4096 (0x1000) possible default store identifiers. The **DEFAULT_STANDARD** represents the range of 4096 possible action values reserved for UEFI-defined default store identifiers. See Section 33.2.5.8 for more information on defaults.
typedef UINTN EFI_BROWSER_ACTION_REQUEST;

#define EFI_BROWSER_ACTION_REQUEST_NONE 0
#define EFI_BROWSER_ACTION_REQUEST_RESET 1
#define EFI_BROWSER_ACTION_REQUEST_SUBMIT 2
#define EFI_BROWSER_ACTION_REQUEST_EXIT 3
#define EFI_BROWSER_ACTION_REQUEST_FORM_SUBMIT_EXIT 4
#define EFI_BROWSER_ACTION_REQUEST_FORM_DISCARD_EXIT 5
#define EFI_BROWSER_ACTION_REQUEST_FORM_APPLY 6
#define EFI_BROWSER_ACTION_REQUEST_FORM_DISCARD 7
#define EFI_BROWSER_ACTION_REQUEST_RECONNECT 8
#define EFI_BROWSER_ACTION_REQUEST_QUESTION_APPLY 9

If the callback function returns with the ActionRequest set to _NONE, then the Forms Browser will take no special behavior.

If the callback function returns with the ActionRequest set to _RESET, then the Forms Browser will exit and request the platform to reset.

If the callback function returns with the ActionRequest set to _SUBMIT, then the Forms Browser will save all modified question values to storage and exit.

If the callback function returns with the ActionRequest set to _EXIT, then the Forms Browser will discard all modified question values and exit.

If the callback function returns with the ActionRequest set to _FORM_SUBMIT_EXIT, then the Forms Browser will write all modified question values on the selected form to storage and then exit the selected form.

If the callback function returns with the ActionRequest set to _FORM_DISCARD_EXIT, then the Forms Browser will discard the modified question values on the selected form and then exit the selected form.

If the callback function returns with the ActionRequest set to _FORM_APPLY, then the Forms Browser will write all modified current question values on the selected form to storage.

If the callback function returns with the ActionRequest set to _FORM_DISCARD, then the Forms Browser will discard the current question values on the selected form and replace them with the original question values.

If the callback function returns with the ActionRequest set to _RECONNECT, a hardware and/or software configuration change was performed by the user, and the controller needs to be reconnected for the driver to recognize the change. Upon the user exiting the formset or the browser, the Forms Browser is required to call the EFI Boot Service DisconnectController() followed by the EFI Boot Service ConnectController() to reconnect the controller. The controller handle passed to DisconnectController() and ConnectController() is the handle on which this EFI_HII_CONFIG_ACCESS_PROTOCOL is installed.

If the callback function returns with the ActionRequest set to _QUESTION_APPLY, then the Forms Browser will write the current modified question value on the selected form to storage.

### Status Codes Returned

<table>
<thead>
<tr>
<th>Code</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>EFI_SUCCESS</td>
<td>The callback successfully handled the action.</td>
</tr>
</tbody>
</table>
35.6 Form Browser Protocol

The **EFI_FORM_BROWSER2_PROTOCOL** is the interface to call for drivers to leverage the EFI configuration driver interface.

**EFI_FORM_BROWSER2_PROTOCOL**

**Summary**

The **EFI_FORM_BROWSER2_PROTOCOL** is the interface to the UEFI configuration driver. This interface will allow the caller to direct the configuration driver to use either the HII database or use the passed-in packet of data.

**GUID**

```c
#define EFI_FORM_BROWSER2_PROTOCOL_GUID \
{ 0xb9d4c360, 0xbcfb, 0x4f9b, \ 
{ 0x92, 0x98, 0x53, 0xc1, 0x36, 0x98, 0x22, 0x58 } }
```

**Protocol Interface Structure**

```c
typedef struct _EFI_FORM_BROWSER2_PROTOCOL {
    EFI_SEND_FORM2 SendForm;
    EFI_BROWSER_CALLBACK2 BrowserCallback;
} EFI_FORM_BROWSER2_PROTOCOL;
```

**Parameters**

- **SendForm**
  
  Browse the specified configuration forms. See the **SendForm()** function description.

- **BrowserCallback**
  
  Routine used to expose internal configuration state of the browser. This is primarily used by callback handler routines which were called by the browser and in-turn need to get additional information from the browser itself. See the **BrowserCallback()** function description.

**Description**

This protocol is the interface to call for drivers to leverage the EFI configuration driver interface.

**EFI_FORM_BROWSER2_PROTOCOL.SendForm()**

**Summary**

Initialize the browser to display the specified configuration forms.
Prototype

```c
typedef
EFI_STATUS
(EFIAPI *EFI_SEND_FORM2) (  
  IN CONST EFI_FORM_BROWSER2_PROTOCOL *This,
  IN EFI_HII_HANDLE *Handles,
  IN UINTN HandleCount,
  IN CONST EFI_GUID *FormsetGuid, OPTIONAL
  IN EFI_FORM_ID FormId, OPTIONAL
  IN CONST EFI_SCREEN_DESCRIPTOR *ScreenWidth, OPTIONAL
  OUT EFI_BROWSER_ACTION_REQUEST *ActionRequest OPTIONAL
);
```

Parameters

**This**
A pointer to the `EFI_FORM_BROWSER2_PROTOCOL` instance.

**Handles**
A pointer to an array of HII handles to display. This value should correspond to the value of the HII form package that is required to be displayed. Type `EFI_HII_HANDLE` is defined in `EFI_HII_DATABASE_PROTOCOL.NewPackageList()` in Section 33.3.1.

**HandleCount**
The number of handles in the array specified by `Handle`.

**FormsetGuid**
This field points to the `EFI_GUID` which must match the `Guid` field or one of the elements of the ClassId field in the `EFI_IFR_FORM_SET` op-code. If `FormsetGuid` is `NULL`, then this function will display the form set class `EFI_HII_PLATFORM_SETUP_FORMSET_GUID`.

**FormId**
This field specifies the identifier of the form within the form set to render as the first displayable page. If this field has a value of 0x0000, then the Forms Browser will render the first enabled form in the form set.

**ScreenWidth**
Points to recommended form dimensions, including any non-content area, in characters. Type `EFI_SCREEN_DESCRIPTOR` is defined in "Related Definitions" below.

**ActionRequested**
Points to the action recommended by the form.

Description
This function is the primary interface to the Forms Browser. The Forms Browser displays the forms specified by `FormsetGuid` and `FormId` from all of HII handles specified by `Handles`. If more than one
form can be displayed, the Forms Browser will provide some means for the user to navigate between the forms in addition to that provided by cross-references in the forms themselves.

If *ScreenDimensions* is non-NULL, then it points to a recommended display size for the form. If browsing in text mode, then these are recommended character positions. If browsing in graphics mode, then these values are converted to pixel locations using the standard font size (8 pixels per horizontal character cell and 19 pixels per vertical character cell). If *ScreenDimensions* is NULL the browser may choose the size based on platform policy. The browser may choose to ignore the size based on platform policy.

If *ActionRequested* is non-NULL, then upon return, it points to an enumerated value (see *EFI_BROWSER_ACTION* in “Related Definitions” below) which describes the action requested by the user. If set to *EFI_BROWSER_ACTION_NONE*, then no specific action was requested by the form. If set to *EFI_BROWSER_ACTION_RESET*, then the form requested that the platform be reset. The browser may, based on platform policy, ignore such action requests.

If *FormsetGuid* is set to *EFI_HII_PLATFORM_SETUP_FORMSET_GUID*, it indicates that the form set contains forms designed to be used for platform configuration. If *FormsetGuid* is set to *EFI_HII_DRIVER_HEALTH_FORMSET_GUID*, it indicates that the form set contains forms designed to be used for support of the Driver Health Protocol (see Section 11.10). If FormsetGuid is set to *EFI_HII_USER_CREDENTIAL_FORMSET_GUID*, it indicates that the form set contains forms designed to be used for support of the User Credential Protocol (see Section 36.3.2). If *FormsetGuid* is set to *EFI_HII_REST_STYLE_FORMSET_GUID*, it indicates that the form set contains forms designed to be used for support configuration of REST architectural style (see Section 29.7) Other values may be used for other applications.

### Related Definitions

```c
typedef struct {
    UINTN LeftColumn;
    UINTN RightColumn;
    UINTN TopRow;
    UINTN BottomRow;
} EFI_SCREEN_DESCRIPTOR;
```

*LeftColumn*  
Value that designates the text column where the browser window will begin from the left-hand side of the screen

*RightColumn*  
Value that designates the text column where the browser window will end on the right-hand side of the screen.

*TopRow*  
Value that designates the text row from the top of the screen where the browser window will start.
typedef UINTN EFI_BROWSER_ACTION_REQUEST;

#define EFI_BROWSER_ACTION_REQUEST_NONE 0
#define EFI_BROWSER_ACTION_REQUEST_RESET 1
#define EFI_BROWSER_ACTION_REQUEST_SUBMIT 2
#define EFI_BROWSER_ACTION_REQUEST_EXIT 3

The value EFI_BROWSER_ACTION_REQUEST_NONE indicates that no specific caller action is required. The value EFI_BROWSER_ACTION_REQUEST_RESET indicates that the caller requested a platform reset. The value EFI_BROWSER_ACTION_REQUEST_SUBMIT indicates that a callback requested that the browser submit all values and exit. The value EFI_BROWSER_ACTION_REQUEST_EXIT indicates that a callback requested that the browser exit without saving all values.

#define EFI_HII_PLATFORM_SETUP_FORMSET_GUID \ 
  { 0x93039971, 0x8545, 0x4b04, \ 
  { 0xb4, 0x5e, 0x32, 0xeb, 0x83, 0x26, 0x04, 0x0e } } 

#define EFI_HII_DRIVER_HEALTH_FORMSET_GUID \ 
  { 0xf22fc20c, 0x8cf4, 0x45eb, \ 
  { 0x8e, 0x06, 0xad, 0x4e, 0x50, 0xb9, 0x5d, 0xd3 } } 

#define EFI_HII_USER_CREDENTIAL_FORMSET_GUID \ 
  { 0x337f4407, 0x5aee, 0x4b83, \ 
  { 0xb2, 0xa7, 0x4e, 0xad, 0xca, 0x30, 0x88, 0xcd } } 

#define EFI_HII_REST_STYLE_FORMSET_GUID \ 
  { 0x790217bd, 0xbecf, 0x485b, \ 
  { 0x91, 0x70, 0x5f, 0xf7, 0x11, 0x31, 0x8b, 0x27 } } 

Status Codes Returned

EFI_SUCCESS The function completed successfully
EFI_NOT_FOUND No valid forms could be found to display.
EFI_INVALID_PARAMETER One of the parameters has an invalid value.

EFI_FORM_BROWSER2_PROTOCOL.BrowserCallback()

Summary
This function is called by a callback handler to retrieve uncommitted state data from the browser.
Prototype

```
EFI_STATUS (EFIAPI *EFI_BROWSER_CALLBACK2 ) ( 
    IN   CONST EFI_FORM_BROWSER2_PROTOCOL   *This,
    IN OUT UINTN    *ResultsDataSize,
    IN OUT EFI_STRING   ResultsData,
    IN BOOLEAN   RetrieveData,
    IN   CONST EFI_GUID   *VariableGuid, OPTIONAL
    IN   CONST CHAR16    *VariableName OPTIONAL
    ) ;
```

Parameters

- **This**
  A pointer to the EFI_FORM_BROWSER2_PROTOCOL instance.

- **ResultsDataSize**
  A pointer to the size of the buffer associated with ResultsData. On input, the size in bytes of ResultsData. On output, the size of data returned in ResultsData.

- **ResultsData**
  A string returned from an IFR browser or equivalent. The results string will have no routing information in them.

- **RetrieveData**
  A BOOLEAN field which allows an agent to retrieve (if RetrieveData = TRUE) data from the uncommitted browser state information or set (if RetrieveData = FALSE) data in the uncommitted browser state information.

- **VariableGuid**
  An optional field to indicate the target variable GUID name to use.

- **VariableName**
  An optional field to indicate the target human-readable variable name.

Description

This service is typically called by a driver's callback routine which was in turn called by the browser. The routine called this service in the browser to retrieve or set certain uncommitted state information that resides in the open formsets.

Status Codes Returned

- **EFI_SUCCESS**
  The results have been distributed or are awaiting distribution.

- **EFI_BUFFER_TOO_SMALL**
  The ResultsDataSize specified was too small to contain the results data.

- **EFI_UNSUPPORTED**
  Uncommitted browser state is not available at the current stage of execution.
35.7 HII Popup Protocol

EFI_HII_POPUP_PROTOCOL

Summary
This protocol provides services to display a popup window.
The protocol is typically produced by the forms browser and consumed by a driver’s callback handler.

GUID
#define EFI_HII_POPUP_PROTOCOL_GUID \
{ 0x4311edc0, 0x6054, 0x46d4, { 0x9e, 0x40, 0x89, 0x3e, 0xa9, 0x52, 0xfc, 0xcc }
}

Protocol Interface Structure
typedef struct {
    UINT64 Revision;
    EFI_HII_CREATE_POPUP CreatePopup;
} EFI_HII_POPUP_PROTOCOL;

Parameters
Revision
Protocol revision
CreatePopup
Displays a popup window

Related Definitions
#define EFI_HII_POPUP_PROTOCOL_REVISION 1

EFI_HII_POPUP_PROTOCOL.CreatePopup()

Summary
Displays a popup window.

Prototype
typedef
EFI_STATUS
(EFIAPI * EFI_HII_CREATE_POPUP) (
    IN EFI_HII_POPUP_PROTOCOL *This,
    IN EFI_HII_POPUP_STYLE PopupStyle,
    IN EFI_HII_POPUP_TYPE PopupType,
    EFI_HII_HANDLE HiiHandle
    IN EFI_STRING_ID Message,
    OUT EFI_HII_POPUP_SELECTION *UserSelectionOPTIONAL,
    );
Parameters

This
A pointer to the \texttt{EFI_HII_POPUP_PROTOCOL} instance.

PopupStyle
Popup style to use. \texttt{EFI_HII_POPUP_STYLE} is defined in the “Related Definitions” below.

PopupType
Type of the popup to display. \texttt{EFI_HII_POPUP_TYPE} is defined in the “Related Definitions” below.

HiiHandle
HII handle of the string pack containing \texttt{Message}.

Message
A message to display in the popup box.

UserSelection
User selection. \texttt{EFI_HII_POPUP_SELECTION} is defined in the “Related Definitions” below.

Description

The \texttt{CreatePopup()} function displays a modal message box that contains string specified by \texttt{Message}. Explicit line break characters can be used to specify a multi-line message (see \texttt{Section 33.2.6.2.4}). A popup window may contain user selectable options. The option selected by a user is returned via an optional \texttt{UserSelection} parameter.

A list of options presented to a user is defined by the \texttt{PopupType}.

The \texttt{PopupStyle} provides a hint to protocol implementation regarding nature of the message being displayed. The function may optionally use \texttt{PopupStyle} to customize visual appearance of the message box.

\texttt{EfiHiiPopupTypeOk} is a simple popup window with a single user selectable option that can be used to acknowledge the message. If \texttt{UserSelection} is specified, it is set to \texttt{EfiHiiPopupSelectionOk}.

\texttt{EfiHiiPopupTypeOkCancel} is a popup window with two user selectable options: OK and Cancel.

\texttt{EfiHiiPopupTypeYesNo} is a popup window with two user selectable options: Yes and No.

\texttt{EfiHiiPopupTypeYesNoCancel} is a popup window with three user selectable options: Yes, No, and Cancel.
Related Definitions

typedef enum {
    EfiHiiPopupStyleInfo,
    EfiHiiPopupStyleWarning,
    EfiHiiPopupStyleError
} EFI_HII_POPUP_STYLE;

typedef enum {
    EfiHiiPopupTypeOk,
    EfiHiiPopupTypeOkCancel,
    EfiHiiPopupTypeYesNo,
    EfiHiiPopupTypeYesNoCancel
} EFI_HII_POPUP_TYPE;

typedef enum {
    EfiHiiPopupSelectionOk,
    EfiHiiPopupSelectionCancel,
    EfiHiiPopupSelectionYes,
    EfiHiiPopupSelectionNo
} EFI_HII_POPUP_SELECTION;

Status Codes Returned

EFI_SUCCESS The popup box was successfully displayed

EFI_INVALID_PARAMETER HiiHandle and Message do not define is a valid HII string.

EFI_INVALID_PARAMETER PopupType is not one of the values defined by this specification.

EFI_OUT_OF_RESOURCES There are not enough resources available to display the popup box.
36 - User Identification

36.1 User Identification Overview

This section describes services which describe the current user of the platform. A user is the entity which is controlling the behavior of the machine. The user may be an individual, a class or group of individuals or another machine.

Each user has a user profile. There is always at least one user profile for a machine. This profile governs the behavior of the user identification process until another user has been selected. The nature and definition of these privileges are beyond the scope of this section. One user profile is always active and describes the platform’s current user.

New user profiles are introduced into the system through enrollment. During enrollment, information about a new user is gathered. Some of this information identifies the user for specific purposes, such as a user’s name or a user’s network domain. Other information is gathered in the form of credentials, which is information which can be used at a later time to verify the identity of a user. Credentials are generally divided into three categories: something you know (password), something you have (smart card, smart token, RFID), something you are (fingerprint). The means by which a platform determines the user’s identity based on credentials is user identification.

In the simplest case, a single set of credentials are required to establish a user’s identity. This is called single-factor authentication. In more rigorous cases, multiple credentials might be required to establish a user’s identity or different privilege levels given if only a single factor is available. This is called multi-factor authentication.

If the credentials are checked only once, this is called static authentication. For example, a sign-on box where the user enters a password and provides a fingerprint would be examples of static authentication. However, if credentials (and thus the user’s identity) can be changed during system execution, this is called dynamic authentication. For example, a smart token which can be hot-removed from the system or an RFID tag which is moved in and out of range would be examples of dynamic authentication.

The user identity manager is the optional UEFI driver which manages the process of determining the user’s identity and storing information about the user.

The user enrollment manager is the optional application which adds or enrolls new users, gathering the necessary information to ascertain their identity in the future.

The credential provider driver manages a single class of credentials. Examples include a USB fingerprint sensor, a smart card or a password. The means by which these drivers are selected and invoked is beyond the scope of this specification.

36.1.1 User Identify

The process of identifying the user occurs after platform initialization has made the services described in the EFI System Table available. Before the Boot Manager behavior described in chapter 3, a user profile must be established. The user profile established might be:

- A default user profile, giving a standard set of privileges. This is similar to a “guest” login.
• A *default* user profile, based on a User Credential Provider where `Default()` returns `AutoLogon = TRUE`.

• A specific user profile, established using the `Identify()` function of the User Manager.

Every time the user profile is modified, the User Identity Manager will signal the `EFI_EVENT_GROUP_USER_PROFILE_CHANGED` event. The current user profile can only be changed by calling the User Identity Manager’s `Identify()` function or as the result of a credential provider calling the `Notify()` function (when dynamic authentication is supported). The `Identify()` function changes the current user profile after examining the credentials provided by the various credential providers and comparing these against those found in the user profile database.

When the UEFI Boot Manager signals the `EFI_EVENT_GROUP_READY_TO_BOOT` event group, the User Identity Manager publishes the current user profile information in the EFI System Configuration Table.

Depending on the security considerations in the implementation (see Section 36.1.4), user identification can continue into different phases of execution.

1. **Boot Manager, Once.** In this scenario, identification is permitted until the `EFI_EVENT_GROUP_READY_TO_BOOT` is signaled by the Boot Manager. After this time, user identification is not allowed again. This is the simplest, since the user profile database can be locked at this time using a simple one-time lock.

2. **Boot Manager, Multiple.** In this scenario, identification is permitted until the `EFI_EVENT_GROUP_READY_TO_BOOT` is signaled by the Boot Manager. After this time, if the boot option returns back into the Boot Manager, identification is allowed again. This scenario requires that the user profile database only be updatable while in the Boot Manager.
3. Until ExitBootServices. In this scenario, identification is permitted until the EFI_EVENT_GROUP_EXIT_BOOT_SERVICES is signaled by the Boot Manager. This scenario requires that the user profile database cannot be updated by unauthorized executables.

36.1.2 User Profiles

The user profiles are collections of information about users. There is always a current user (and thus, a currently selected user profile). The user profiles are stored in a user profile database.

Each user profile has the following attributes:

§ User Identifier

User identifiers are unique to a particular user profile. The uniqueness of the user profile identifier must persist across reboots. Credentials return this identifier during the identification process.

§ User Identification Policy

The user identification policy determines which credentials must be presented in order to establish the user’s identity and set the user profile as the current user profile. The policy consists of a boolean expression consisting of credential handles and the operators AND, OR and NOT. This allows the user profile to be selected, for example, depending on a password credential OR a fingerprint credential. Or the profile might be selected depending on a password credential AND a fingerprint credential.

§ User Privileges

The user privileges control what the user can and cannot do. For example, can the user enroll other users, boot off of a selected device, etc.

§ User Information

User information consists of typed data records attached to the user profile handle. Some of this information is non-volatile. Some of this information may be provided by a specific credential driver. User information is classified as public, private or protected:

• Public user information is available at any time.
• Private user information is only available while it is part of the current user profile.
• Protected user information is only available once user has been authenticated by a credential provider.

Drivers and applications can be notified when the current user profile is changed, by using the UEFI Boot Service CreateEventEx() and the EFI_EVENT_GROUP_USER_PROFILE_CHANGED

User profiles are available while the User Identity Manager is running, but only the current user profile is available after the UEFI Boot Manager has started execution.
36.1.2.1 User Profile Database

The user profile database is a repository of all users known to the user identity manager. The user profile database should be maintained in non-volatile memory and this memory must be protected against corruption and erasure.

The user profile database is considered “open” if the user profile database can still be updated and the current profile can still be changed using the EFI User Manager Protocol. The user profile database is considered “closed” if the user profile database cannot be updated nor the current user profile changes using the EFI User Manager Protocol.

36.1.2.2 User Identification Policy

The user identification policy is a boolean expression which determines which class of credential or which credential providers must assert the user’s identity in order to a user profile to be eligible for selection as the current user profile.

For example, assume that you want a password:

\[\text{CredentialClass(Password)}\]

This expression would assert true if any credential provider asserts that a user has successfully entered a password.

\[\text{CredentialClass(Password)} \&\& \text{CredentialClass(Fingerprint)}\]

This expression would require the user to present both a fingerprint AND a password in order to select this user profile.

\[\text{CredentialClass(Password)} \mid\| \text{CredentialClass(Fingerprint)}\]

This expression, on the other hand, allows the user to present a fingerprint OR a password in order to select this user profile.

Let’s say you only want the Phoenix password provider:

\[\text{CredentialClass(Password)} \&\& \text{CredentialProvider(Phoenix)}\]

In all of these cases, the class of credential and the provider of the credential are actually GUIDs. The standard credential class GUIDs are assigned by this specification. The credential provider identifiers are generated by the companies creating the credential providers.

36.1.3 Credential Providers

The User Credential Provider drivers follow the UEFI driver model. During initialization, they install an instance of the EFI Driver Binding Protocol. For hardware devices, the User Credential Provider may consume the bus I/O protocol and produce the User Credential Protocol. For software-based User Credential Providers, the User Credential Protocol could be installed on the image handler. The exact implementation depends on the number of separate credential types that the User Identity Manager will display.

When \text{Start()} is called, they:
1. Install one instance of the `EFI_USER_CREDENTIAL2_PROTOCOL` for each simultaneous user which might be authenticated. For example, if more than one smart token were inserted, then one instance might be created for each token. However, for a fingerprint sensor, one instance might be created for all fingerprint sensors managed by the same driver.

2. Install the user-interface forms used for interacting with the user using the HII Database Protocol. The form must be encoded using the GUID `EFI_USER_CREDENTIAL2_PROTOCOL_GUID`.

3. Install the EFI HII Configuration Access Protocol to handle interaction with the UEFI forms browser. This protocol is called to retrieve the current information from the credential provider. It is also called when the user presses OK to save the current form values. It also provides the callback functionality which allows real-time processing of the form values.

User Credential Providers are responsible to creating a one-to-one mapping between a device, fingerprint or password and a user identifier.

This specification does not explicitly support passing of user credential information related to operating system logon to an OS-present environment. For example, User Credential Providers may encrypt the user credential information and pass it, either as a part of the User Information Table or the EFI System Configuration Table, to an OS-present driver or application.

This specification does not explicitly support OS-present update of user credential information or user identification policy. Such support may be implemented in many ways, including the usage of write-authenticated EFI variables (see `SetVariable()` or capsules (see `UpdateCapsule()`).

### 36.1.4 Security Considerations

Since the current profile details a number of security-related privileges, it is important that the User Identity Manager and User Credential Providers and the environment in which they execute are trusted.

This includes:

- Protecting the storage area where these drivers are stored.
- Protecting the means by which these drivers are selected.
- Protecting the execution environment of these drivers from unverified drivers.
- The data structures used by these drivers should not be corrupted by unauthorized drivers while they are still being used.
In many cases, the User Identity Manager, the User Credential drivers and the on-board drivers are located in a protected location (e.g. a write-protected flash device) and the platform policy for these locations allows them to be trusted.

However, other drivers may be loaded from unprotected location or may be loaded from devices (such as PCI cards) or a hard drive which are easily replaced. Therefore, all drivers loaded prior to the User Identity Manager should be verified. No unverified drivers or applications should be allowed to execute while decisions based on the current user policy are still being made.

For example, either the default platform policy must successfully be able to verify drivers listed in the \texttt{Driver\\#\\#} load options, or else the user must be identified prior to processing these drivers. Otherwise, the drivers’ execution should be deferred. If the user profile is changed through a subsequent call to \texttt{Identify()} or through dynamic authentication, the \texttt{Driver\\#\\#} options may not be processed again.

In systems where the user profile database and current user profile can be protected from corruption, the user profile database is closed when the system signals the event \texttt{EFI_EXIT_BOOT_SERVICES\_EVENT\_GUID}. In systems where the user profile database and current user profile cannot be protected from corruption, the user profile database is closed when the system signals the event \texttt{EFI\_READY\_TO\_BOOT\_EVENT\_GUID}. 

### 36.1.5 Deferred Execution

The platform may need to defer the execution of an image because of security considerations. For example, see LoadImage(). Information about the images which are not executed because of security considerations may be recorded and then reported by installing an instance of the `EFI_DEFERRED_IMAGE_LOAD_PROTOCOL` (see Section 36.3.3). There may be more than one producer of the protocol.

The firmware’s boot manager may use the instances of this protocol in order to automatically load drivers whose execution was deferred because of inadequate privileges once the current user profile contains adequate security privileges.

This boot manager can reevaluate the deferred images each time that the event `EFI_EVENT_GROUP_USER.IDENTITY_CHANGED` is signaled.

Images which have been loaded may not be unloaded when the current user profile is changed, even if the new user profile would have prevented that driver from being loaded.

### 36.2 User Identification Process

This section describes the typical initialization steps required for the user identification process.

#### 36.2.1 User Identification Process

1. The User Identity Manager is launched. This driver reads all of the user profiles from the user profile database, sets the default user profile as the current profile, and installs an instance of the `EFI_USER_MANAGER_PROTOCOL`.
2. Each credential provider driver registers their user-interface related forms and installs an instance of the `EFI_USER_CREDENTIAL2_PROTOCOL`.
3. The User Identity Manager’s `Identify()` function is called to select the current user.
4. The User Identity Manager enumerates all of the User Credential Providers required by the User Identification Policy:
   a. Select the User Credential Provider which returns `Default = TRUE` from the `Default()` function. If more than one return TRUE or none return TRUE, choose a default based on implementation-specific criteria (last-logged-on, etc.)
   b. If that credential provider also returns `AutoLogon = TRUE` from the `Default()` function, then call `User()`. If no error was returned and a user profile with the specified user identifier exists, select the specified user profile as the current user profile and jump to step 9.
5. The User Identity Manager enumerates all of the User Credential Providers required by the User Identification Policy:
   a. Call the `Title()` and (optionally) the `Tile()` function to retrieve the text and image indicated for each User Credential Provider.
   b. Call the `Form()` function to retrieve the form indicated for each User Credential Provider.
   c. Create the user interface to allow the user to select between the different User Credential Providers.
Highlight the default User Credential Provider, as specified in step 4.a.

6. If the user selects one of the User Credential Providers, call `Select()`. If `AutoLogon = TRUE` on return, then call `User()`. If no error was returned and a user profile with the specified user identifier exists, select the specified user profile as the current user profile and jump to step 9.

7. Interact with the user. Regular interaction can occur using the `Callback()` functions. If another User Credential Provider is selected then `Deselect()` is called for the current User Credential Provider and `Select()` is called for the newly selected User Credential Provider.

8. If the user presses OK then the User Manager will saved settings using the EFI Configuration Access protocol. Then it will call the `User()` function of each credential provider. If it returns successfully and one of the user policies evaluates to true, then select the specified user profile as the current user profile and go to step 9. Otherwise display an error and go back.

9. Go through all of the credential providers using `GetNextInfo()` and `GetInfo()` and add the information to the current user profile.

10. Exit

### 36.2.2 Changing The Current User Profile

This section describes the typical actions taken when the current user profile is changed.

1. If there was already a valid current user profile, then all records marked as `private` in that profile are no longer available.
2. All records marked as `private` in the new user profile will be available.
3. The handle of the current user profile is changed.
4. An event with the GUID `EFI_EVENT_GROUP_USER_IDENTITY_CHANGED` is signaled to indicate that the current user profile has been changed.

### 36.2.3 Ready To Boot

Before the boot manager is read to pass control to the boot option and signals the `EFI_EVENT_GROUP_READY_TO_BOOT` event group, the User Identity Manager will publish the current user profile into the EFI System Configuration Table with the `EFI_USER_MANAGER_PROTOCOL_GUID`. The format is described in the User Information Table (Section 36.5). It will also save all non-volatile profile information.

User Credential drivers with non-volatile storage should also store non-volatile credential information which has changed.
36.3 Code Definitions

36.3.1 User Manager Protocol

**EFI_USER_MANAGER_PROTOCOL**

**Summary**

Reports information about a user.

**GUID**

```c
#define EFI_USER_MANAGER_PROTOCOL_GUID \
{ 0x6fd5b00c, 0xd426, 0x4283, \ 
{ 0x98, 0x87, 0x6c, 0xf5, 0xcf, 0x1c, 0xb1, 0xfe } };
```

**Protocol Interface Structure**

```c
typedef struct _EFI_USER_MANAGER_PROTOCOL {
    EFI_USER_PROFILE_CREATE Create;
    EFI_USER_PROFILE_DELETE Delete;
    EFI_USER_PROFILE_GET_NEXT GetNext;
    EFI_USER_PROFILE_CURRENT Current;
    EFI_USER_PROFILE_IDENTIFY Identify;
    EFI_USER_PROFILE_FIND Find;
    EFI_USER_PROFILE_NOTIFY Notify;
    EFI_USER_PROFILE_GET_INFO GetInfo;
    EFI_USER_PROFILE_SET_INFO SetInfo;
    EFI_USER_PROFILE_DELETE_INFO DeleteInfo;
    EFI_USER_PROFILE_GET_NEXT_INFO GetNextInfo;
} EFI_USER_MANAGER_PROTOCOL;
```

**Parameters**

- **Create**
  Create a new user profile.

- **Delete**
  Delete an existing user profile.

- **GetNext**
  Cycle through all user profiles.

- **Current**
  Return the current user profile.

- **Identify**
  Identify a user and set the current user profile using credentials.

- **Find**
  Find a user by a piece of user information.
Notify

Notify the user manager driver that credential information has changed.

GetInfo

Return information from a user profile.

SetInfo

Change information in a user profile.

DeleteInfo

Delete information from a user profile.

GetNextInfo

Cycle through all information from a user profile.

Description

This protocol manages user profiles.

**EFI_USER_MANAGER_PROTOCOL.Create()**

**Summary**

Create a new user profile.

**Prototype**

```c
typedef EFI_STATUS
    (EFIAPI *EFI_USER_PROFILE_CREATE) (
    IN CONST EFI_USER_MANAGER_PROTOCOL *This,
    OUT EFI_USER_PROFILE_HANDLE *User
    );
```

**Parameters**

*This

Points to this instance of the **EFI_USER_MANAGER_PROTOCOL**.

*User

On return, points to the new user profile handle. The user profile handle is unique only during this boot.

**Description**

This function creates a new user profile with only a new user identifier attached and returns its handle. The user profile is non-volatile, but the handle *User can change across reboots.

If the current user profile does not permit creation of new user profiles then **EFI_ACCESS_DENIED** will be returned. If creation of new user profiles is not supported, then **EFI_UNSUPPORTED** is returned.
Related Definitions

typedef VOID *EFI_USER_PROFILE_HANDLE;

Status Codes Returned

<table>
<thead>
<tr>
<th>Status Code</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>EFI_SUCCESS</td>
<td>User profile was successfully created.</td>
</tr>
<tr>
<td>EFI_ACCESS_DENIED</td>
<td>Current user does not have sufficient permissions to create a user profile.</td>
</tr>
<tr>
<td>EFI_UNSUPPORTED</td>
<td>Creation of new user profiles is not supported.</td>
</tr>
<tr>
<td>EFI_INVALID_PARAMETER</td>
<td>User is NULL.</td>
</tr>
</tbody>
</table>

EFI_USER_MANAGER_PROTOCOL.Delete()

Summary
Delete an existing user profile.

Prototype

```
typedef
EFI_STATUS
(EIFIAPI *EFI_USER_PROFILE_DELETE) (                          
  IN CONST EFI_USER_MANAGER_PROTOCOL *This,
  IN EFI_USER_PROFILE_HANDLE User
);
```

Parameters

- **This**
  Points to this instance of the EFI_USER_MANAGER_PROTOCOL.
- **User**
  User profile handle. Type EFI_USER_PROFILE_HANDLE is defined in Create().

Description
Delete an existing user profile. If the current user profile does not permit deletion of user profiles then EFI_ACCESS_DENIED will be returned. If there is only a single user profile then EFI_ACCESS_DENIED will be returned. If deletion of user profiles is not supported, then EFI_UNSUPPORTED will be returned.

Status Codes Returned

<table>
<thead>
<tr>
<th>Status Code</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>EFI_SUCCESS</td>
<td>User profile was successfully deleted.</td>
</tr>
<tr>
<td>EFI_ACCESS_DENIED</td>
<td>Current user does not have sufficient permissions to delete a user profile or there is only one user profile.</td>
</tr>
<tr>
<td>EFI_UNSUPPORTED</td>
<td>Deletion of new user profiles is not supported.</td>
</tr>
<tr>
<td>EFI_INVALID_PARAMETER</td>
<td>User does not refer to a valid user profile.</td>
</tr>
</tbody>
</table>
EFI_USER_MANAGER_PROTOCOL.GetNext()

Summary
Enumerate all of the enrolled users on the platform.

Prototype

```c
typedef EFI_STATUS (EFIAPI *EFI_USER_PROFILE_GET_NEXT)(
    IN   CONST EFI_USER_MANAGER_PROTOCOL *This,
    IN OUT EFI_USER_PROFILE_HANDLE *User
);
```

Parameters

- **This**
  Points to the instance of this EFI_USER_MANAGER_PROTOCOL.

- **User**
  On entry, points to the previous user profile handle or NULL to start enumeration. On exit, points to the next user profile handle or NULL if there are no more user profiles.

Description

This function returns the next enrolled user profile. To retrieve the first user profile handle, point User at a NULL. Each subsequent call will retrieve another user profile handle until there are no more, at which point User will point to NULL.

**Note:** There is always at least one user profile.

Status Codes Returned

<table>
<thead>
<tr>
<th>Status Code</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>EFI_SUCCESS</td>
<td>Next enrolled user profile successfully returned.</td>
</tr>
<tr>
<td>EFI_INVALID_PARAMETER</td>
<td>User is NULL.</td>
</tr>
<tr>
<td>EFI_ACCESS_DENIED</td>
<td>Next enrolled user profile was not successfully returned.</td>
</tr>
</tbody>
</table>

EFI_USER_MANAGER_PROTOCOL.Current()

Summary
Return the current user profile handle.
Prototype

typedef
EFI_STATUS
(EIFIAPI *EFI_USER_PROFILE_CURRENT)(
    IN CONST EFI_USER_MANAGER_PROTOCOL *This,
    OUT EFI_USER_PROFILE_HANDLE *CurrentUser
);

Parameters

    This

    Points to this instance of the EFI_USER_MANAGER_PROTOCOL.

    CurrentUser

    On return, points to the current user profile handle.

Description

This function returns the current user profile handle.

Status Codes Returned

<table>
<thead>
<tr>
<th>Code</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>EFI_SUCCESS</td>
<td>Current user profile handle returned successfully.</td>
</tr>
<tr>
<td>EFI_INVALID_PARAMETER</td>
<td>CurrentUser is NULL.</td>
</tr>
</tbody>
</table>

EFI_USER_MANAGER_PROTOCOL.Identify()

Summary

Identify a user.

Prototype

typedef
EFI_STATUS
(EIFIAPI *EFI_USER_IDENTIFY) (
    IN   CONST EFI_USER_MANAGER_PROTOCOL *This,
    OUT  EFI_USER_PROFILE_HANDLE *User
);

Parameters

    This

    Points to the instance of the EFI_USER_MANAGER_PROTOCOL.

    User

    On return, points to the user profile handle for the current user profile.

Description

Identify the user and, if authenticated, returns the user handle and changes the current user profile.
All user information marked as private in a previously selected profile is no longer available for inspection.

Whenever the current user profile is changed then the an event with the GUID ` EFI_EVENT_GROUP_USER_PROFILE_CHANGED` is signaled.

The function can only be called at `TPL_APPLICATION`.

Related Definitions

```c
#define EFI_EVENT_GROUP_USER_PROFILE_CHANGED \  { 0xbaf1e6de, 0x209e, 0x4adb, \  { 0x8d, 0x96, 0xfd, 0x8b, 0x71, 0xf3, 0xf6, 0x83 } }
```

Status Codes Returned

<table>
<thead>
<tr>
<th>Status Code</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>EFI_SUCCESS</td>
<td>User was successfully identified.</td>
</tr>
<tr>
<td>EFI_INVALID_PARAMETER</td>
<td>User is NULL.</td>
</tr>
<tr>
<td>EFI_ACCESS_DENIED</td>
<td>User was not successfully identified.</td>
</tr>
</tbody>
</table>

EFI_USER_MANAGER_PROTOCOL.Find()

Summary

Find a user using a user information record.

Prototype

```c
typedef EFI_STATUS
(EFI_API *EFI_USER_PROFILE_FIND)(
    IN   CONST EFI_USER_MANAGER_PROTOCOL *This,
    IN OUT EFI_USER_PROFILE_HANDLE *User,
    IN OUT EFI_USER_INFO_HANDLE *UserInfo OPTIONAL,
    IN   CONST EFI_USER_INFO *Info,
    IN   UINTN               InfoSize
);
```

Parameters

`This`

Points to this instance of the `EFI_USER_MANAGER_PROTOCOL`.

`User`

On entry, points to the previously returned user profile handle or `NULL` to start searching with the first user profile. On return, points to the user profile handle or `NULL` if not found.
**UserInfo**

On entry, points to the previously returned user information handle or **NULL** to start searching with the first. On return, points to the user information handle of the user information record or **NULL** if not found. Can be **NULL**, in which case only one user information record per user can be returned. Type **EFI_USER_INFO_HANDLE** is defined in “Related Definitions” below.

**Info**

Points to the buffer containing the user information to be compared to the user information record. If the user information record data is empty, then only the user information record type is compared.

If **InfoSize** is 0, then the user information record data must be empty.

**InfoSize**

The size of **Info**, in bytes.

**Description**

This function searches all user profiles for the specified user information record. The search starts with the user information record handle following **UserInfo** and continues until either the information is found or there are no more user profiles.

A match occurs when the **Info.InfoType** field matches the user information record type and the user information record data matches a portion of **Info**.

**Status Codes Returned**

<table>
<thead>
<tr>
<th>Status Code</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>EFI_SUCCESS</td>
<td>User information was found. <strong>User</strong> points to the user profile handle and <strong>UserInfo</strong> points to the user information handle.</td>
</tr>
<tr>
<td>EFI_NOT_FOUND</td>
<td>User information was not found. <strong>User</strong> points to NULL and <strong>UserInfo</strong> points to NULL.</td>
</tr>
<tr>
<td>EFI_INVALID_PARAMETER</td>
<td><strong>User</strong> is <strong>NULL</strong>. Or <strong>Info</strong> is <strong>NULL</strong>.</td>
</tr>
</tbody>
</table>

**Related Definitions**

```c
typedef VOID *EFI_USER_INFO_HANDLE;
```

**EFI_USER_MANAGER_PROTOCOL.Notify()**

**Summary**

Called by credential provider to notify of information change.
Prototype

typedef
EFI_STATUS
(EIFIAPI *EFI_USER_PROFILE_NOTIFY)(
    IN CONST EFI_USER_MANAGER_PROTOCOL *This,
    IN EFI_HANDLE Changed
);

Parameters

This
Points to this instance of the EFI_USER_MANAGER_PROTOCOL.

Changed
Handle on which is installed an instance of the EFI_USER_CREDENTIAL2_PROTOCOL where the user has changed.

Description

This function allows the credential provider to notify the User Identity Manager when user status has changed.

If the User Identity Manager doesn’t support asynchronous changes in credentials, then this function should return EFI_UNSUPPORTED.

If current user does not exist, and the credential provider can identify a user, then make the user to be current user and signal the EFI_EVENT_GROUP_USER_PROFILE_CHANGED event.

If current user already exists, and the credential provider can identify another user, then switch current user to the newly identified user, and signal the EFI_EVENT_GROUP_USER_PROFILE_CHANGED event.

If current user was identified by this credential provider and now the credential provider cannot identify current user, then logout current user and signal the EFI_EVENT_GROUP_USER_PROFILE_CHANGED event.

Status Codes Returned

<table>
<thead>
<tr>
<th>Status Code</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>EFI_SUCCESS</td>
<td>The User Identity Manager has handled the notification.</td>
</tr>
<tr>
<td>EFI_NOT_READY</td>
<td>The function was called while the specified credential provider was not selected.</td>
</tr>
<tr>
<td>EFI_UNSUPPORTED</td>
<td>The User Identity Manager doesn’t support asynchronous notifications.</td>
</tr>
</tbody>
</table>
 EFI_USER_MANAGER_PROTOCOL.GetInfo()

Summary
Return information attached to the user.

Prototype

typedef
   EFI_STATUS
   (EFIAPI *EFI_USER_PROFILE_GET_INFO)(
      IN   CONST EFI_USER_MANAGER_PROTOCOL *This,
      IN   EFI_USER_PROFILE_HANDLE       User,
      IN   EFI_USER_INFO_HANDLE         UserInfo,
      OUT  EFI_USER_INFO               *Info,
      IN OUT UINTN  *InfoSize
   );

Parameters

This
Points to this instance of the EFI_USER_MANAGER_PROTOCOL.

User
Handle of the user whose profile will be retrieved.

UserInfo
Handle of the user information data record. Type EFI_USER_INFO_HANDLE is defined in GetInfo().

Info
On entry, points to a buffer of at least *InfoSize bytes. On exit, holds the user information. If the buffer is too small to hold the information, then EFI_BUFFER_TOO_SMALL is returned and InfoSize is updated to contain the number of bytes actually required. Type EFI_USER_INFO is described in “Related Definitions” below.

InfoSize
On entry, points to the size of Info. On return, points to the size of the user information.

Description
This function returns user information. The format of the information is described in User Information. The function may return EFI_ACCESS_DENIED if the information is marked private and the handle specified by User is not the current user profile. The function may return EFI_ACCESS_DENIED if the information is marked protected and the information is associated with a credential provider for which the user has not been authenticated.
Status Codes Returned

<table>
<thead>
<tr>
<th>Status Code</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>EFI_SUCCESS</td>
<td>Information returned successfully.</td>
</tr>
<tr>
<td>EFI_ACCESS_DENIED</td>
<td>The information about the specified user cannot be accessed by the current user.</td>
</tr>
<tr>
<td>EFI_BUFFER_TOO_SMALL</td>
<td>The number of bytes specified by *InfoSize is too small to hold the returned data. The actual size required is returned in *InfoSize.</td>
</tr>
<tr>
<td>EFI_NOT_FOUND</td>
<td>User does not refer to a valid user profile or UserInfo does not refer to a valid user info handle.</td>
</tr>
<tr>
<td>EFI_INVALID_PARAMETER</td>
<td>Info is NULL or InfoSize is NULL</td>
</tr>
</tbody>
</table>

Related Definitions

```c
typedef struct {
  EFI_GUID              Credential;
  UINT8                 InfoType;
  UINT8                 Reserved1;
  EFI_USER_INFO_ATTRIBS InfoAttribs;
  UINT32                InfoSize;
} EFI_USER_INFO;
```

*Credential*

The user credential identifier associated with this user information or else Nil if the information is not associated with any specific credential.

*InfoType*

The type of user information. See `EFI_USER_INFO_x_RECORD` in User Information for a description of the different types of user information.

*Reserved1*

Must be set to 0.

*InfoAttribs*

The attributes of the user profile information.

*InfoSize*

The size of the user information, in bytes, including this header.
typedef UINT16 EFI_USER_INFO_ATTRIBS;

#define EFI_USER_INFO_STORAGE        0x000F
#define EFI_USER_INFO_STORAGE_VOLATILE   0x0000
#define EFI_USER_INFO_STORAGE_CREDENTIAL_NV 0x0001
#define EFI_USER_INFO_STORAGE_PLATFORM_NV  0x0002

#define EFI_USER_INFO_ACCESS        0x0070
#define EFI_USER_INFO_PUBLIC        0x0010
#define EFI_USER_INFO_PRIVATE        0x0020
#define EFI_USER_INFO_PROTECTED       0x0030
#define EFI_USER_INFO_EXCLUSIVE       0x0080

The EFI_USER_INFO_STORAGE_x values describe how the user information should be stored. If EFI_USER_INFO_STORAGE_VOLATILE is specified, then the user profile information will be lost after a reboot. If EFI_USER_INFO_STORAGE_CREDENTIAL_NV is specified, then the information will be stored by the driver which created the handle Credential. If USER_INFO_STORAGE_PLATFORM_NV is specified, then the information will be stored by the User Identity Manager in platform non-volatile storage.

There are three levels of access to information associated with the user profile: public, private or protected. If EFI_USER_INFO_PUBLIC is specified, then the user profile information is available always. If EFI_USER_INFO_PRIVATE is specified, then the user profile information is only available if the user has been authenticated (whether or not they are the current user). If EFI_USER_INFO_PROTECTED is specified, then the user profile information is only available if the user has been authenticated and is the current user.

If EFI_USER_INFO_EXCLUSIVE is specified then there can only be one user information record of this type in the user profile. Attempts to use SetInfo() will fail.

EFI_USER_MANAGER_PROTOCOL.SetInfo()

Summary
Add or update user information.

Prototype
typedef

EFI_STATUS
(EFI_API *EFI_USER_PROFILE_SET_INFO) (  
IN   CONST EFI_USER_MANAGER_PROTOCOL *This,  
IN   EFI_USER_PROFILE_HANDLE User,  
IN OUT EFI_USER_INFO_HANDLE       *UserInfo,  
IN   CONST EFI_USER_INFO *Info,  
IN    UINTN                InfoSize
);
Parameters

This
Points to this instance of the \texttt{EFI\_USER\_MANAGER\_PROTOCOL}.

User
Handle of the user whose profile will be changed.

UserInfo
On entry, points to the handle of the user information record to change or \texttt{NULL} if the user information should be added to the user profile. On exit, points to the handle of the user credential information record.

Info
Points to the user information. See \texttt{EFI\_USER\_INFO} for more information.

InfoSize
The size of \texttt{Info}, in bytes.

Description
This function changes user information. If \texttt{NULL} is pointed to by \texttt{UserInfo}, then a new user information record is created and its handle is returned in \texttt{UserInfo}. Otherwise, the existing one is replaced.

If \texttt{EFI\_USER\_INFO\_IDENTITY\_POLICY\_RECORD} is changed, it is the caller's responsibility to keep it to be synced with the information on credential providers.

If \texttt{EFI\_USER\_INFO\_EXCLUSIVE} is specified in \texttt{Info} and a user information record of the same type already exists in the user profile, then \texttt{EFI\_ACCESS\_DENIED} will be returned and \texttt{UserInfo} will point to the handle of the existing record.

Status Codes Returned

<table>
<thead>
<tr>
<th>Status Code</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>\texttt{EFI_SUCCESS}</td>
<td>User profile information was successfully changed/added.</td>
</tr>
<tr>
<td>\texttt{EFI_ACCESS_DENIED}</td>
<td>The record is exclusive.</td>
</tr>
<tr>
<td>\texttt{EFI_SECURITY_VIOLATION}</td>
<td>The current user does not have permission to change the specified user profile or user information record.</td>
</tr>
<tr>
<td>\texttt{EFI_NOT_FOUND}</td>
<td>User does not refer to a valid user profile or \texttt{UserInfo} does not refer to a valid user info handle.</td>
</tr>
<tr>
<td>\texttt{EFI_INVALID_PARAMETER}</td>
<td>\texttt{Info} is \texttt{NULL} or \texttt{InfoSize} is \texttt{NULL}</td>
</tr>
</tbody>
</table>
 EFI_USER_MANAGER_PROTOCOL.DeleteInfo()

Summary
Delete user information.

Prototype

```c
typedef EFI_STATUS
  (EFIAPI *EFI_USER_PROFILE_DELETE_INFO) (  
    IN   CONST EFI_USER_MANAGER_PROTOCOL   *This,
    IN   EFI_USER_PROFILE_HANDLE          User,
    IN   EFI_USER_INFO_HANDLE            UserInfo
  );
```

Parameters

- **This**
  Points to this instance of the EFI_USER_MANAGER_PROTOCOL.

- **User**
  Handle of the user whose information will be deleted.

- **UserInfo**
  Handle of the user information to remove.

Description
Delete the user information attached to the user profile specified by the UserInfo.

Status Codes Returned

<table>
<thead>
<tr>
<th>Status Code</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>EFI_SUCCESS</td>
<td>User information deleted successfully.</td>
</tr>
<tr>
<td>EFI_NOT_FOUND</td>
<td>User information record UserInfo does not exist in the user profile.</td>
</tr>
<tr>
<td>EFI_ACCESS_DENIED</td>
<td>The current user does not have permission to delete this user information.</td>
</tr>
</tbody>
</table>

EFI_USER_MANAGER_PROTOCOL.GetNextInfo()

Summary
Enumerate all of the enrolled users on the platform.

Prototype

```c
typedef
EFI_STATUS
(EIFIAPI *EFI_USER_PROFILE_GET_NEXT_INFO)(
  IN   CONST EFI_USER_MANAGER_PROTOCOL *This,
  IN   EFI_USER_PROFILE_HANDLE User,
  IN OUT EFI_USER_INFO_HANDLE      *UserInfo
);
```

Parameters

This
Points to the instance of this EFI_USER_MANAGER_PROTOCOL.

User
Handle of the user whose information will be enumerated

UserInfo
On entry, points to the previous user information handle or NULL to start enumeration. On exit, points to the next user information handle or NULL if there is no more user information.

Description
This function returns the next user information record. To retrieve the first user information record handle, point UserInfo at a NULL. Each subsequent call will retrieve another user information record handle until there are no more, at which point UserInfo will point to NULL.

Status Codes Returned

<table>
<thead>
<tr>
<th>Status Code</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>EFI_SUCCESS</td>
<td>User information returned.</td>
</tr>
<tr>
<td>EFI_NOT_FOUND</td>
<td>No more user information found.</td>
</tr>
<tr>
<td>EFI_INVALID_PARAMETER</td>
<td>UserInfo is NULL.</td>
</tr>
</tbody>
</table>
36.3.2 Credential Provider Protocols

**EFI_USER_CREDENTIAL2_PROTOCOL**

**Summary**
Provide support for a single class of credentials

**GUID**

```c
#define EFI_USER_CREDENTIAL2_PROTOCOL_GUID \ 
{ 0xe98adb03, 0xb8b9, 0x4af8, \ 
{ 0xba, 0x20, 0x26, 0xe9, 0x11, 0x4c, 0xbc, 0xe5 } }
```

**Prototype**

```c
typedef struct _EFI_USER_CREDENTIAL2_PROTOCOL {
    EFI_GUID Identifier;
    EFI_GUID Type;
    EFI_CREDENTIAL_ENROLL Enroll;
    EFI_CREDENTIAL_FORM Form;
    EFI_CREDENTIAL_TILE Tile;
    EFI_CREDENTIAL_TITLE Title;
    EFI_CREDENTIAL_USER User;
    EFI_CREDENTIAL_SELECT Select;
    EFI_CREDENTIAL_DESELECT Deselect;
    EFI_CREDENTIAL_DEFAULT Default;
    EFI_CREDENTIAL_GET_INFO GetInfo;
    EFI_CREDENTIAL_GET_NEXT_INFO GetNextInfo;
    EFI_CREDENTIAL_CAPABILITIES Capabilities;
    EFI_CREDENTIAL_DELETE Delete;
} EFI_USER_CREDENTIAL2_PROTOCOL;
```

**Parameters**

- **Identifier**
  Uniquely identifies this credential provider.

- **Type**
  Identifies this class of User Credential Provider. See **EFI_CREDENTIAL_CLASS_x** in “Related Definitions” below.

- **Enroll**
  Enroll a user using this credential provider.

- **Form**
  Return the form set and form identifier for the form.

- **Tile**
  Returns an optional bitmap image used to identify this credential provider.
Title

Returns a string used to identify this credential provider.

User

Returns the user profile identifier ascertained by using this credential.

Select

Called when a credential provider is selected.

Deselect

Called when a credential provider is deselected.

Default

Returns whether the credential provider can provide the default credential.

GetInfo

Return user information provided by the credential provider.

GetNextInfo

Cycle through all user information available from the credential provider.

Capabilities

Bitmask which describes the capabilities supported by the credential provider. Type EFI_CREDENTIAL_CAPABILITIES is defined in “Related Definitions” below.

Delete

Delete a user on this credential provider.

Description

Attached to a device handle, this protocol identifies a single means of identifying the user.

If EFI_CREDENTIAL_CAPABILITIES_ENROLL is specified, then this credential provider supports the ability to enroll new user identification information using the Enroll() function.

Related Definitions

#define EFI_USER_CREDENTIAL_CLASS_UNKNOWN \
 { 0x5cf32e68, 0x7660, 0x449b, \
   { 0x80, 0xe6, 0x7e, 0xa3, 0x6e, 0x3, 0xf6, 0xa8 } }; 

#define EFI_USER_CREDENTIAL_CLASS_PASSWORD \
 { 0xf8e5058c, 0xccb6, 0x4714, \
   { 0xb2, 0x20, 0x3f, 0x7e, 0x3a, 0x64, 0xb, 0xd1 } }; 

#define EFI_USER_CREDENTIAL_CLASS_SMART_CARD \
 { 0x5f03ba33, 0x8c6b, 0x4c24, \
   { 0xaa, 0x2e, 0x14, 0xa2, 0x65, 0x64, 0xb, 0xd4, 0x54 } }; 

#define EFI_USER_CREDENTIAL_CLASS_FINGERPRINT \
 { 0x32cba21f, 0xf308, 0x4cbc, \
   { 0x9a, 0xb5, 0xf5, 0xa2, 0x65, 0x69, 0x9f, 0x4, 0x4a } };
#define EFI_USER_CREDENTIAL_CLASS_HANDPRINT \
{ 0x5917ef16, 0xf723, 0x4bb9, \ 
{ 0xa6, 0x4b, 0xd8, 0xc5, 0x32, 0xf4, 0xd8, 0xb5 } }; 

#define EFI_USER_CREDENTIAL_CLASS_SECURE_CARD \
{ 0x8a6b4a83, 0x42fe, 0x45d2, \
{ 0xa2, 0xef, 0x46, 0xf0, 0x6c, 0x7d, 0x98, 0x52 } }; 

typedef UINT64 EFI_CREDENTIAL_CAPABILITIES; 

#define EFI_CREDENTIAL_CAPABILITIES_ENROLL 0x0000000000000001 

EFI_USER_CREDENTIAL2_PROTOCOL.Enroll()

Summary
Enroll a user on a credential provider.

Prototype
typedef

EFI_STATUS

(EFIAP2 *EFI_CREDENTIAL2_ENROLL)(
  IN CONST EFI_USER_CREDENTIAL2_PROTOCOL *This,
  IN EFI_USER_PROFILE_HANDLE User
); 

Parameters
This
Points to this instance of the EFI_USER_CREDENTIAL2_PROTOCOL.

User
The user profile to enroll.

Description
This function enrolls a user on this credential provider. If the user exists on this credential provider, update the user information on this credential provider; otherwise add the user information on credential provider.
**Status Codes Returned**

<table>
<thead>
<tr>
<th>Code</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>EFI_SUCCESS</td>
<td>User profile was successfully enrolled</td>
</tr>
<tr>
<td>EFI_ACCESS_DENIED</td>
<td>Current user profile does not permit enrollment on the user profile handle. Either the user profile cannot enroll on any user profile or cannot enroll on a user profile other than the current user profile.</td>
</tr>
<tr>
<td>EFI_UNSUPPORTED</td>
<td>This credential provider does not support enrollment in the pre-OS.</td>
</tr>
<tr>
<td>EFI_DEVICE_ERROR</td>
<td>The new credential could not be created because of a device error.</td>
</tr>
<tr>
<td>EFI_INVALID_PARAMETER</td>
<td>User does not refer to a valid user profile handle.</td>
</tr>
</tbody>
</table>

**EFI_USER_CREDENTIAL2_PROTOCOL.Form()**

**Summary**

Returns the user interface information used during user identification.

**Prototype**

```c
typedef EFI_STATUS
    (EFIAPI *EFI_CREDENTIAL_Form)(
    IN CONST EFI_USER_CREDENTIAL2_PROTOCOL *This,
    OUT EFI_HII_HANDLE *Hii,
    OUT EFI_GUID *FormSetId,
    OUT EFI_FORM_ID *FormId
    );
```

**Parameters**

- **This**
  Points to this instance of the `EFI_USER_CREDENTIAL2_PROTOCOL`.

- **Hii**
  On return, holds the HII database handle. Type `EFI_HII_HANDLE` is defined in `EFI_HII_DATABASE_PROTOCOL.NewPackageList()` in the Packages section.

- **FormSetId**
  On return, holds the identifier of the form set which contains the form used during user identification.

- **FormId**
  On return, holds the identifier of the form used during user identification.

**Description**

This function returns information about the form used when interacting with the user during user identification. The form is the first enabled form in the form-set class `EFI_HII_USER_CREDENTIAL_FORMSET_GUID` installed on the HII handle `HiiHandle`. If the user credential provider does not require a form to identify the user, then this function should return `EFI_NOT_FOUND`. 
Status Codes Returned

<table>
<thead>
<tr>
<th>Code</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>EFI_SUCCESS</td>
<td>Form returned successfully.</td>
</tr>
<tr>
<td>EFI_NOT_FOUND</td>
<td>Form not returned.</td>
</tr>
<tr>
<td>EFI_INVALID_PARAMETER</td>
<td><code>Hii</code> is <code>NULL</code> or <code>FormSetId</code> is <code>NULL</code> or <code>FormId</code> is <code>NULL</code></td>
</tr>
</tbody>
</table>

**EFI_USER_CREDENTIAL2_PROTOCOL.Tile()**

**Summary**

Returns bitmap used to describe the credential provider type.

**Prototype**

```c
typedef
EFI_STATUS
(EIFIAP *EFI_CREDENTIAL_TILE)(
    IN CONST EFI_USER_CREDENTIAL2_PROTOCOL *This,
    IN OUT UINTN       *Width,
    IN OUT UINTN       *Height,
    OUT  EFI_HII_HANDLE  *Hii,
    OUT  EFI_IMAGE_ID   *Image
);
```

**Parameters**

- **This**
  Points to this instance of the `EFI_USER_CREDENTIAL2_PROTOCOL`.

- **Width**
  On entry, points to the desired bitmap width. If `NULL` then no bitmap information will be returned. On exit, points to the width of the bitmap returned.

- **Height**
  On entry, points to the desired bitmap height. If `NULL` then no bitmap information will be returned. On exit, points to the height of the bitmap returned.

- **Hii**
  On return, holds the HII database handle. Type `EFI_HII_HANDLE` is defined in `EFI_HII_DATABASE_PROTOCOL.NewPackageList()` in the Packages section.

- **Image**
  On return, holds the HII image identifier. Type `EFI_IMAGE_ID` is defined in this specification, Section 34.4.

**Description**

This optional function returns a bitmap which is less than or equal to the number of pixels specified by `Width` and `Height`. If no such bitmap exists, then `EFI_NOT_FOUND` is returned.
Status Codes Returned

<table>
<thead>
<tr>
<th>Status Code</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>EFI_SUCCESS</td>
<td>Image identifier returned successfully.</td>
</tr>
<tr>
<td>EFI_NOT_FOUND</td>
<td>Image identifier not returned.</td>
</tr>
<tr>
<td>EFI_INVALID_PARAMETER</td>
<td>Hii is NULL or Image is NULL.</td>
</tr>
</tbody>
</table>

**EFI_USER_CREDENTIAL2_PROTOCOL.Title()**

**Summary**
Returns string used to describe the credential provider type.

**Prototype**
```c
typedef EFI_STATUS
  (EFIAPI *EFI_CREDENTIAL2_PROTOCOL)(
    IN   CONST EFI_USER_CREDENTIAL2_PROTOCOL *This,
    OUT  EFI_HII_HANDLE   *Hii,
    OUT  EFI_STRING_ID    *String
  );
```

**Parameters**
- **This**
  Points to this instance of the **EFI_USER_CREDENTIAL2_PROTOCOL**.
- **Hii**
  On return, holds the HII database handle. TType **EFI_HII_HANDLE** is defined in **EFI_HII_DATABASE_PROTOCOL.NewPackageList()** in the Packages section.
- **String**
  On return, holds the HII string identifier. Type **EFI_STRING_ID** is defined in **Section 33.3.8.2.1**.

**Description**
This function returns a string which describes the credential provider. If no such string exists, then **EFI_NOT_FOUND** is returned.

**Status Codes Returned**

<table>
<thead>
<tr>
<th>Status Code</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>EFI_SUCCESS</td>
<td>String identifier returned successfully.</td>
</tr>
<tr>
<td>EFI_NOT_FOUND</td>
<td>String identifier not returned.</td>
</tr>
<tr>
<td>EFI_INVALID_PARAMETER</td>
<td>Hii is NULL or String is NULL.</td>
</tr>
</tbody>
</table>

**EFI_USER_CREDENTIAL2_PROTOCOL.User()**

**Summary**
Return the user identifier associated with the currently authenticated user.
Prototype

```c
typedef
  EFI_STATUS
  (EFIAPIC *EFI_CREDENTIAL_USER)(
    IN   CONST EFI_USER_CREDENTIAL2_PROTOCOL *This,
    IN   EFI_USER_PROFILE_HANDLE User,
    OUT  EFI_USER_INFO_IDENTIFIER *Identifier
  );
```

Parameters

- **This**
  Points to this instance of the `EFI_USER_CREDENTIAL2_PROTOCOL`.

- **User**
  The user profile handle of the user profile currently being considered by the user identity manager. If `NULL`, then no user profile is currently under consideration.

- **Identifier**
  On return, points to the user identifier. Type `EFI_USER_INFO_IDENTIFIER` is defined in “Related Definitions” below.

Description

This function returns the user identifier of the user authenticated by this credential provider. This function is called after the credential-related information has been submitted on a form OR after a call to `Default()` has returned that this credential is ready to log on.

This function can return one of five possible responses:

- If no user profile can yet be identified, then `EFI_NOT_READY` is returned.
- If the user has been locked out, then `EFI_ACCESS_DENIED` is returned.
- If the user specified by `User` is identified, then Identifier returns with the user identifier associated with that handle and `EFI_SUCCESS` is returned.
- If `Identifier` is `NULL`, then `EFI_INVALID_PARAMETER` is returned.
- If specified `User` does not refer to a valid user profile, then `EFI_NOT_FOUND` is returned.

Status Codes Returned

<table>
<thead>
<tr>
<th>Code</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>EFI_SUCCESS</td>
<td>User identifier returned successfully.</td>
</tr>
<tr>
<td>EFI_NOT_READY</td>
<td>No user identifier can be returned.</td>
</tr>
<tr>
<td>EFI_ACCESS_DENIED</td>
<td>The user has been locked out of this user credential.</td>
</tr>
<tr>
<td>EFI_NOT_FOUND</td>
<td><code>User</code> is not <code>NULL</code>, and the specified user handle can't be found in user profile database</td>
</tr>
<tr>
<td>EFI_INVALID_PARAMETER</td>
<td>Identifier is <code>NULL</code>.</td>
</tr>
</tbody>
</table>
**EFI_USER_CREDENTIAL2_PROTOCOL.Select()**

**Summary**
Indicate that user interface interaction has begun for the specified credential.

**Prototype**
```c
typedef EFI_STATUS
  (EFIAPI *EFI_CREDENTIAL_SELECT)(
    IN CONST EFI_USER_CREDENTIAL2_PROTOCOL *This,
    OUT EFI_CREDENTIAL_LOGON_FLAGS *AutoLogon
  );
```

**Parameters**
- **This**
  Points to this instance of the `EFI_USER_CREDENTIAL2_PROTOCOL`.
- **AutoLogon**
  On return, points to the credential provider’s capabilities after the credential provider has been selected by the user. Type `EFI_CREDENTIAL_LOGON_FLAGS` is defined in “Related Definitions” below.

**Description**
This function is called when a credential provider is selected by the user. If `AutoLogon` returns `FALSE`, then the user interface will be constructed by the User Identity Manager.

**Related Definitions**
```c
typedef UINT32 EFI_CREDENTIAL_LOGON_FLAGS;
#define EFI_CREDENTIAL_LOGON_FLAG_AUTO  0x00000001
#define EFI_CREDENTIAL_LOGON_FLAG_DEFAULT 0x00000002
```

If `EFI_CREDENTIAL_LOGON_FLAG_AUTO` is set, then the User Identity Manager may use this as a hint to try logging on immediately. If not set, then the User Identity Manager may use this as an indication to wait for the user to submit the information.

If `EFI_CREDENTIAL_LOGON_FLAG_DEFAULT` is set, then the User Identity Manager may use this as a hint to use this credential provider as the default credential provider. If more than one credential provider returns with this set, then the selection is implementation specific. If `EFI_CREDENTIAL_LOGON_FLAG_DEFAULT` is set and `EFI_CREDENTIAL_LOGON_FLAG_AUTO` is set then the User Identity Manager may use this as a hint to log the user on immediately.

**Status Codes Returned**

<table>
<thead>
<tr>
<th>Code</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>EFI_SUCCESS</td>
<td>Credential provider successfully selected.</td>
</tr>
<tr>
<td>EFI_INVALID_PARAMETER</td>
<td><code>AutoLogon</code> is <code>NULL</code></td>
</tr>
</tbody>
</table>
**EFI_USER_CREDENTIAL2_PROTOCOL.Deselect()**

**Summary**
Indicate that user interface interaction has ended for the specified credential.

**Prototype**
```c
typedef EFI_STATUS
    (EFIAPI *EFI_CREDENTIAL_DESELECT)(
    IN CONST EFI_USER_CREDENTIAL2_PROTOCOL *This
    );
```

**Parameters**
- **This**
  Points to this instance of the EFI_USER_CREDENTIAL2_PROTOCOL.

**Description**
This function is called when a credential provider is deselected by the user.

**Status Codes Returned**

<table>
<thead>
<tr>
<th>Status Code</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>EFI_SUCCESS</td>
<td>Credential provider successfully selected.</td>
</tr>
</tbody>
</table>

**EFI_USER_CREDENTIAL2_PROTOCOL.Default()**

**Summary**
Return the default logon behavior for this user credential.

**Prototype**
```c
typedef EFI_STATUS
    (EFIAPI *EFI_CREDENTIAL_DEFAULT)(
    IN CONST EFI_USER_CREDENTIAL2_PROTOCOL *This,
    OUT EFI_CREDENTIAL_LOGON_FLAGS *AutoLogon
    );
```

**Parameters**
- **This**
  Points to this instance of the EFI_USER_CREDENTIAL2_PROTOCOL.
- **AutoLogon**
  On return, holds whether the credential provider should be used by default to automatically log on the user. Type EFI_CREDENTIAL_LOGON_FLAGS is defined in EFI_USER_CREDENTIAL2_PROTOCOL.Select().
Description
This function reports the default login behavior regarding this credential provider.

Status Codes Returned

<table>
<thead>
<tr>
<th>Status Code</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>EFI_SUCCESS</td>
<td>Default information successfully returned.</td>
</tr>
<tr>
<td>EFI_INVALID_PARAMETER</td>
<td>AutoLogon is NULL</td>
</tr>
</tbody>
</table>

**EFI_USER_CREDENTIAL2_PROTOCOL.GetInfo()**

Summary
Return information attached to the credential provider.

Prototype

```c
typedef
   EFI_STATUS
   (EFIAPIC*EFI_CREDENTIAL_GET_INFO)(
   IN   CONST EFI_USER_CREDENTIAL2_PROTOCOL *This,
   IN   EFI_USER_INFO_HANDLE UserInfo,
   OUT   EFI_USER_INFO *Info,
   IN OUT UINTN *InfoSize
   );
```

Parameters

*This*  
Points to this instance of the **EFI_USER_CREDENTIAL2_PROTOCOL**.

User Info  
Handle of the user information data record. Type **EFI_USER_INFO_HANDLE** is defined in **GetInfo()**.

*Info*  
On entry, points to a buffer of at least *InfoSize bytes. On exit, holds the user information. If the buffer is too small to hold the information, then **EFI_BUFFER_TOO_SMALL** is returned and *InfoSize* is updated to contain the number of bytes actually required. Type **EFI_USER_INFO** is described in “Related Definitions” below.

*InfoSize*  
On entry, points to the size of *Info. On return, points to the size of the user information.

Description
This function returns user information.
Status Codes Returned

<table>
<thead>
<tr>
<th>Status Code</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>EFI_SUCCESS</td>
<td>Information returned successfully.</td>
</tr>
<tr>
<td>EFI_BUFFER_TOO_SMALL</td>
<td>The size specified by InfoSize is too small to hold all of the user information. The size required is returned in *InfoSize.</td>
</tr>
<tr>
<td>EFI_NOT_FOUND</td>
<td>The specified UserInfo does not refer to a valid user info handle.</td>
</tr>
<tr>
<td>EFI_INVALID_PARAMETER</td>
<td>Info is NULL or InfoSize is NULL.</td>
</tr>
</tbody>
</table>

EFI_USER_CREDENTIAL2_PROTOCOL.GetNextInfo()

Summary
Enumerate all of the user information records on the credential provider..

Prototype

```c
typedef EFI_STATUS (EFIAPI *EFI_USER_CREDENTIAL2_PROTOCOL_GET_NEXT_INFO)(
  IN   CONST EFI_USER_CREDENTIAL2_PROTOCOL *This,
  IN OUT EFI_USER_INFO_HANDLE *UserInfo
);
```

Parameters

*This
Points to the instance of this EFI_USER_CREDENTIAL2_PROTOCOL.

*UserInfo
On entry, points to the previous user information handle or NULL to start enumeration. On exit, points to the next user information handle or NULL if there is no more user information.

Description
This function returns the next user information record. To retrieve the first user information record handle, point *UserInfo at a NULL. Each subsequent call will retrieve another user information record handle until there are no more, at which point *UserInfo will point to NULL.

Status Codes Returned

<table>
<thead>
<tr>
<th>Status Code</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>EFI_SUCCESS</td>
<td>User information returned.</td>
</tr>
<tr>
<td>EFI_NOT_FOUND</td>
<td>No more user information found.</td>
</tr>
<tr>
<td>EFI_INVALID_PARAMETER</td>
<td>*UserInfo is NULL.</td>
</tr>
</tbody>
</table>

EFI_USER_CREDENTIAL2_PROTOCOL.Delete()

Summary
Delete a user on a credential provider.
Prototype

```c
typedef
EFI_STATUS (EFIAPI *EFI_CREDENTIAL_DELETE)(
    IN CONST EFI_USER_CREDENTIAL2_PROTOCOL *This,
    IN EFI_USER_PROFILE_HANDLE User);
```

Parameters

This

Points to this instance of the `EFI_USER_CREDENTIAL2_PROTOCOL`.

User

The user profile handle to delete.

Description

This function deletes a user on this credential provider.

Status Codes Returned

<table>
<thead>
<tr>
<th>Status Code</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>EFI_SUCCESS</td>
<td>User profile was successfully deleted.</td>
</tr>
<tr>
<td>EFI_ACCESS_DENIED</td>
<td>Current user profile does not permit deletion on the user profile handle. Either the user profile cannot delete on any user profile or cannot delete on a user profile other than the current user profile.</td>
</tr>
<tr>
<td>EFI_UNSUPPORTED</td>
<td>This credential provider does not support deletion in the pre-OS.</td>
</tr>
<tr>
<td>EFI_DEVICE_ERROR</td>
<td>The new credential could not be deleted because of a device error.</td>
</tr>
<tr>
<td>EFI_INVALID_PARAMETER</td>
<td>User does not refer to a valid user profile handle.</td>
</tr>
</tbody>
</table>

36.3.3 Deferred Image Load Protocol

**EFI_DEFERRED_IMAGE_LOAD_PROTOCOL**

Summary

Enumerates images whose load was deferred due to security considerations.

GUID

```c
#define EFI_DEFERRED_IMAGE_LOAD_PROTOCOL_GUID \    
{ 0x15853d7c, 0x3ddf, 0x43e0, \    
{ 0xa1, 0xcb, 0xeb, 0xf8, 0x5b, 0x8f, 0x87, 0x2c } };
```

Protocol Interface Structure

```c
typedef struct _EFI_DEFERRED_IMAGE_LOAD_PROTOCOL {
    EFI_DEFERRED_IMAGE_INFO GetImageInfo();
} EFI_DEFERRED_IMAGE_LOAD_PROTOCOL;
```
Members

GetImageInfo

Return information about a single deferred image. See GetImageInfo() for more information.

Description

This protocol returns information about images whose load was denied because of security considerations. This information can be used by the Boot Manager or another agent to reevaluate the images when the current security profile has been changed, such as when the current user profile changes. There can be more than one instance of this protocol installed.

 EFI_DEFERRED_IMAGE_LOAD_PROTOCOL.GetImageInfo()

Summary

Returns information about a deferred image.

Prototype

typedef
EFI_STATUS
(EIFIAPIC *EFI_DEFERRED_IMAGE_INFO)(
   IN EFI_DEFERRED_IMAGE_LOAD_PROTOCOL *This,
   IN UINTN ImageIndex,
   OUT EFI_DEVICE_PATH_PROTOCOL **ImageDevicePath,
   OUT VOID **Image,
   OUT UINTN *ImageSize,
   OUT BOOLEAN *BootOption
);

Parameters

This

Points to this instance of the EFI_DEFERRED_IMAGE_LOAD_PROTOCOL.

ImageIndex

Zero-based index of the deferred index.

ImageDevicePath

On return, points to a pointer to the device path of the image. The device path should not be freed by the caller.

Image

On return, points to the first byte of the image or NULL if the image is not available. The image should not be freed by the caller unless LoadImage() has been called successfully.

ImageSize

On return, the size of the image, or 0 if the image is not available.
BootOption

On return, points to **TRUE** if the image was intended as a boot option or **FALSE** if it was not intended as a boot option.

**Description**

This function returns information about a single deferred image. The deferred images are numbered consecutively, starting with 0. If there is no image which corresponds to **ImageIndex**, then **EFI_NOT_FOUND** is returned. All deferred images may be returned by iteratively calling this function until **EFI_NOT_FOUND** is returned.

**Image** may be **NULL** and **ImageSize** set to 0 if the decision to defer execution was made because of the location of the executable image rather than its actual contents.

**Status Codes Returned**

<table>
<thead>
<tr>
<th>Code</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>EFI_SUCCESS</td>
<td>Image information returned successfully.</td>
</tr>
<tr>
<td>EFI_NOT_FOUND</td>
<td>ImageIndex does not refer to a valid image.</td>
</tr>
<tr>
<td>EFI_INVALID_PARAMETER</td>
<td>ImageDevicePath is <strong>NULL</strong> or Image is <strong>NULL</strong> or ImageSize is <strong>NULL</strong> or BootOption is <strong>NULL</strong></td>
</tr>
</tbody>
</table>

### 36.4 User Information

This section describes the different user information and the format of the data. Each of the following records is prefixed with the **EFI_USER_INFO** structure. The format of the record is determined by the type specified by the **InfoType** field in the structure, as listed in the table below:

**Table 36-1 Record values and descriptions**

<table>
<thead>
<tr>
<th>Name</th>
<th>Value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>EFI_USER_INFO_EMPTY_RECORD</td>
<td>0x00</td>
<td>No information.</td>
</tr>
<tr>
<td>EFI_USER_INFO_NAME_RECORD</td>
<td>0x01</td>
<td>User’s name</td>
</tr>
<tr>
<td>EFI_USER_INFO_CREATE_DATE_RECORD</td>
<td>0x02</td>
<td>Date which the user profile was created.</td>
</tr>
<tr>
<td>EFI_USER_INFO_USAGE_DATE_RECORD</td>
<td>0x03</td>
<td>Date which the user profile was last modified.</td>
</tr>
<tr>
<td>EFI_USER_INFO_USAGE_COUNT_RECORD</td>
<td>0x04</td>
<td>Number of times the credential has been used.</td>
</tr>
<tr>
<td>EFI_USER_INFO_IDENTIFIER_RECORD</td>
<td>0x05</td>
<td>User’s unique identifier *</td>
</tr>
<tr>
<td>EFI_USER_INFO_CREDENTIAL_TYPE_RECORD</td>
<td>0x06</td>
<td>Credential type.</td>
</tr>
<tr>
<td>EFI_USER_INFO_CREDENTIAL_TYPE_NAME_RECORD</td>
<td>0x07</td>
<td>Credential type name.</td>
</tr>
<tr>
<td>EFI_USER_INFO_CREDENTIAL_PROVIDER_RECORD</td>
<td>0x08</td>
<td>Credential provider</td>
</tr>
<tr>
<td>EFI_USER_INFO_CREDENTIAL_PROVIDER_NAME_RECORD</td>
<td>0x09</td>
<td>Credential provider name</td>
</tr>
<tr>
<td>EFI_USER_INFO_PKCS11_RECORD</td>
<td>0x0A</td>
<td>PKCS11 Data Object</td>
</tr>
</tbody>
</table>
36.4.1 EFI_USER_INFO_ACCESS_POLICY_RECORD

Summary
Provides the user’s pre-OS access rights.

Prototype

```
#define EFI_USER_INFO_ACCESS_POLICY_RECORD 0x0E

typedef EFI_USER_INFO_ACCESS_CONTROL EFI_USER_INFO_ACCESS_POLICY;
```

Description
This structure described the access policy for the user. There can be, at most, one access policy record per credential (including `NULL` credential). Policy records with a credential specified means that the policy is associated specifically with the credential.

The policy is detailed in a series of encapsulated records of type `EFI_USER_INFO_ACCESS_CONTROL`.

Related Definitions

```
typedef struct {
    UINT32 Type;
    UINT32 Size;
} EFI_USER_INFO_ACCESS_CONTROL;
```

<table>
<thead>
<tr>
<th>EFI_USER_INFO_CBEFF_RECORD</th>
<th>0x0B</th>
<th>ISO 19785 (Common Biometric Exchange Formats Framework) Data Object</th>
</tr>
</thead>
<tbody>
<tr>
<td>EFI_USER_INFO_FAR_RECORD</td>
<td>0x0C</td>
<td>How exact a match is required for biometric identification, measured in percentage.</td>
</tr>
<tr>
<td>EFI_USER_INFO_RETRY_RECORD</td>
<td>0x0D</td>
<td>Number of retries allowed during verification.</td>
</tr>
<tr>
<td>EFI_USER_INFO_ACCESS_POLICY_RECORD</td>
<td>0x0E</td>
<td>Access control information.</td>
</tr>
<tr>
<td>EFI_USER_INFO_IDENTITY_POLICY_RECORD</td>
<td>0x0F</td>
<td>User identity expression.</td>
</tr>
<tr>
<td>EFI_USER_INFO_GUID_RECORD</td>
<td>0xFF</td>
<td>Extended profile information, qualified by a GUID in the header.</td>
</tr>
</tbody>
</table>

Type
Specifies the type of user access control. See `EFI_USER_INFO_ACCESS_x` for more information.

Size
Specifies the size of the user access control record, in bytes, including this header.
36.4.1.1 EFI_USER_INFO_ACCESS_FORBID_LOAD

Summary
Forbids the user from booting or loading executables from the specified device path or any child device paths.

Prototype
```
#define EFI_USER_INFO_ACCESS_FORBID_LOAD  0x00000001
```

Description
This record prohibits the user from loading any executables from zero or device paths or any child device paths. The device paths may contain a specific executable name, in which case the prohibition applies to only that executable.

The record is a series of normal UEFI device paths (not multi-instance device paths).

This prohibition is overridden by the EFI_USER_INFO_ACCESS_PERMIT_LOAD record.

36.4.1.2 EFI_USER_INFO_ACCESS_PERMIT_LOAD

Summary
Permits the user from booting or loading executables from the specified device path or any child device paths.

Prototype
```
#define EFI_USER_INFO_ACCESS_PERMIT_LOAD  0x00000002
```

Description
This record allows the user to load executables from locations specified by zero or more device paths or child paths. The device paths may contain specific executable names, in which case, the permission applies only to that executable.

The record is a series of normal UEFI device paths (not multi-instance device paths).

This prohibition overrides any restrictions put in place by the EFI_USER_INFO_ACCESS_FORBID_LOAD record.

36.4.1.3 EFI_USER_INFO_ACCESS_ENROLL_SELF

Summary
Presence of this record indicates that a user can update enrollment information.

Prototype
```
#define EFI_USER_INFO_ACCESS_ENROLL_SELF 0x00000003
```

Description
If this record is present, then the pre-OS environment will allow the user to initiate an update of authentication information for his/her own profile, but not other user information or other user’s information. This would allow, for example, fingerprint update or password change.
There is no data for this record.

36.4.1.4 EFI_USER_INFO_ACCESS_ENROLL_OTHERS

Summary
Presence of this record indicates that a user can enroll new users.

Prototype

```
#define EFI_USER_INFO_ACCESS_ENROLL_OTHERS 0x00000004
```

Description
If this record is present, then the pre-OS environment will allow the user to initiate enrollment of new user profiles. It does not give permission to update existing user profiles.

There is no data for this record.

36.4.1.5 EFI_USER_INFO_ACCESS_MANAGE

Summary
Presence of this record indicates that a user can update the user information of any user.

Prototype

```
#define EFI_USER_INFO_ACCESS_MANAGE 0x00000005
```

Description
If this record is present, then the pre-OS environment will allow the user to update any information about his/her own profile or other profiles.

There is no data for this record.

36.4.1.6 EFI_USER_INFO_ACCESS_SETUP

Summary
Describes permissions usable when configuring the platform.

Prototype

```
#define EFI_USER_INFO_ACCESS_SETUP 0x00000006
```

Description
This record describes access permission for use in configuring the platform using an UEFI Forms Processor using zero or more GUIDs. There are three standard values (see below) and any number of others may be added.

<table>
<thead>
<tr>
<th>EFI_USER_INFO_ACCESS_SETUP_ADMIN_GUID</th>
<th>System administrator only.</th>
</tr>
</thead>
<tbody>
<tr>
<td>EFI_USER_INFO_ACCESS_SETUP_NORMAL_GUID</td>
<td>Normal user.</td>
</tr>
<tr>
<td>EFI_USER_INFO_ACCESS_SETUP_RESTRICTED_GUID</td>
<td>Restricted user.</td>
</tr>
</tbody>
</table>
Related Definitions

#define EFI_USER_INFO_ACCESS_SETUP_ADMIN_GUID \
{ 0x85b75607, 0xf7ce, 0x471e, \
 { 0xb7, 0xe4, 0x2a, 0xea, 0x5f, 0x72, 0x32, 0xee } };

#define EFI_USER_INFO_ACCESS_SETUP_NORMAL_GUID \
{ 0x1db29ae0, 0x9dcb, 0x43bc, \
{ 0x8d, 0x87, 0x5d, 0xa1, 0x49, 0x64, 0xdd, 0xe2 } };

#define EFI_USER_INFO_ACCESS_SETUP_RESTRICTED_GUID \
{ 0xbd38125, 0x4d63, 0x49f4, \
 { 0x82, 0x12, 0x61, 0xc5, 0xa5, 0x19, 0x0a, 0xf8 } };

36.4.1.7 EFI_USER_INFO_ACCESS_FORBID_CONNECT

Summary
Forbids UEFI drivers from being started from the specified device path(s) or any child device paths.

Prototype
#define EFI_USER_INFO_ACCESS_FORBID_CONNECT  0x00000007

Description
This record prohibits UEFI drivers from being started from the specified device path(s) or any of their child device path(s). This is enforced in the ConnectController() function.

This record prohibits the user from loading a device driver associated with zero or more device paths or their child paths.

The record is a series of normal UEFI device paths (not multi-instance device paths).

This prohibition is overridden by the EFI_USER_INFO_ACCESS_PERMIT_CONNECT record.

36.4.1.8 EFI_USER_INFO_ACCESS_PERMIT_CONNECT

Summary
Permits UEFI drivers to be started on the specified device path(s) or any child device paths.

Prototype
#define EFI_USER_INFO_ACCESS_PERMIT_CONNECT  0x00000008

Description
This record allows loading of device drivers associated with zero or more device paths or their child paths.

The record is a series of normal UEFI device paths (not multi-instance device paths).

This prohibition overrides any restrictions put in place by the EFI_USER_INFO_ACCESS_FORBID_CONNECT record.
36.4.1.9 EFI_USER_INFO_ACCESS_BOOT_ORDER

Summary
Modifies the boot order.

Prototype

```c
#define EFI_USER_INFO_ACCESS_BOOT_ORDER 0x00000009

typedef UINT32 EFI_USER_INFO_ACCESS_BOOT_ORDER_HDR;

#define EFI_USER_INFO_ACCESS_BOOT_ORDER_MASK   0x000F
#define EFI_USER_INFO_ACCESS_BOOT_ORDER_INSERT  0x0000
#define EFI_USER_INFO_ACCESS_BOOT_ORDER_APPEND  0x0001
#define EFI_USER_INFO_ACCESS_BOOT_ORDER_REPLACE 0x0002

#define EFI_USER_INFO_ACCESS_BOOT_ORDER_NODEFAULT 0x0010
```

Description
This exclusive record allows the user profile to insert new boot options at the beginning of the boot order
(EFI_USER_INFO_ACCESS_BOOT_ORDER_INSERT), append new boot options to the end of the boot order
(EFI_USER_INFO_ACCESS_BOOT_ORDER_APPEND) or replace the entire boot order
(EFI_USER_INFO_ACCESS_BOOT_ORDER_REPLACE). If
EFI_USER_INFO_ACCESS_BOOT_ORDER_NODEFAULT is specified then the Boot Manager will not
attempt find a default boot device when the default boot order is does not lead to a bootable device.

The boot options specified by this record are still subject to the permissions specified by
EFI_USER_INFO_ACCESS_FORBID_LOAD and EFI_USER_INFO_ACCESS_PERMIT_LOAD.

The record consists of a single EFI_USER_INFO_ACCESS_BOOT_ORDER_HDR followed by zero or more
UEFI device paths.

36.4.2 EFI_USER_INFO_CBEFF_RECORD

Summary
Provides standard biometric information in the format specified by the ISO 19785 (Common Biometric
Exchange Formats Framework) specification.

Prototype

```c
#define EFI_USER_INFO_CBEFF_RECORD 0x0B

typedef VOID *EFI_USER_INFO_CBEFF;
```

36.4.3 EFI_USER_INFO_CREATE_DATE_RECORD

Summary
Provides the date and time when the user profile was created.
Prototype

```c
#define EFI_USER_INFO_CREATE_DATE_RECORD 0x02
typedef EFI_TIME EFI_USER_INFO_CREATE_DATE;
```

Description
The optional record describing the date and time when the user profile was created. Type `EFI_TIME` is defined in `GetTime()` in this specification.

36.4.4 EFI_USER_INFO_CREDENTIAL_PROVIDER_RECORD

Summary
Specifies the credential provider.

Prototype

```c
#define EFI_USER_INFO_CREDENTIAL_PROVIDER_RECORD 0x08
typedef EFI_GUID EFI_USER_INFO_CREDENTIAL_PROVIDER;
```

Description
This record specifies the credential provider via a unique GUID. The credential’s handle is found in the `EFI_USER_INFO` structure associated with this user information record.

36.4.5 EFI_USER_INFO_CREDENTIAL_PROVIDER_NAME_RECORD

Summary
Specifies the user-readable name of a particular credential’s provider.

Prototype

```c
#define EFI_USER_INFO_CREDENTIAL_PROVIDER_NAME_RECORD 0x09
typedef CHAR16 *EFI_USER_INFO_CREDENTIAL_PROVIDER_NAME;
```

Description
This record specifies the null-terminated name of a particular credential provider. The credential’s handle is found in the `EFI_USER_INFO` structure associated with this user information record.

36.4.6 EFI_USER_INFO_CREDENTIAL_TYPE_RECORD

Summary
Specifies the type of a particular credential associated with the user profile.

Prototype

```c
#define EFI_USER_INFO_CREDENTIAL_TYPE_RECORD 0x06
typedef EFI_GUID EFI_USER_INFO_CREDENTIAL_TYPE;
```

Description
This record specifies the type of a particular credential. The credential’s identifier is found in the Credential field of the EFI_USER_INFO structure. The credential types are listed with the EFI_USER_CREDENTIAL2_PROTOCOL.

36.4.7 EFI_USER_INFO_CREDENTIAL_TYPE_NAME_RECORD

Summary
Specifies the user-readable name of a particular credential type.

Prototype

```c
#define EFI_USER_INFO_CREDENTIAL_TYPE_NAME_RECORD 0x07
typedef CHAR16 *EFI_USER_INFO_CREDENTIAL_TYPE_NAME;
```

Description
This record specifies the null-terminated name of a particular credential type. The credential’ handle is found in the EFI_USER_INFO structure associated with this user information record.

36.4.8 EFI_USER_INFO_GUID_RECORD

Summary
Provides placeholder for additional user profile information identified by a GUID.

Prototype

```c
#define EFI_USER_INFO_GUID_RECORD 0xFF
typedef EFI_GUID EFI_USER_INFO_GUID;
```

Description
This record type provides extensibility by prefixing further data fields in the record with a GUID which identifies the format.

36.4.9 EFI_USER_INFO_FAR_RECORD

Summary
Indicates how close of a match the fingerprint must be in order to be considered a match.

Prototype

```c
#define EFI_USER_INFO_FAR_RECORD 0x0C
typedef UINT8 EFI_USER_INFO_FAR;
```

Description
This record specifies how accurate the fingerprint template match must be in order to be considered a match, as a percentage from 0 (no match) to 100 (perfect match). The accuracy may be for all fingerprint sensors (EFI_USER_INFO.Credential is zero) or for a particular fingerprint sensor (EFI_USER_INFO.Credential is non-zero).
36.4.10 EFI_USER_INFO_IDENTIFIER_RECORD

Summary
Provides a unique non-volatile user identifier for each enrolled user.

Prototype

```c
#define EFI_USER_INFO_IDENTIFIER_RECORD 0x05
typedef UINT8 EFI_USER_INFO_IDENTIFIER[16];
```

Description
The user identifier is unique to each enrolled user and non-volatile. Each user profile must have exactly one of these user information records installed. The format of the value is not specified.

Access
Exclusive:Yes
Modify:Only with user-enrollment permissions.
Visibility:Public.

36.4.11 EFI_USER_INFO_IDENTITY_POLICY_RECORD

Summary
Provides the expression which determines which credentials are required to assert user identity.

Prototype

```c
#define EFI_USER_INFO_IDENTITY_POLICY_RECORD 0x0F
typedef struct {
    UINT32 Type;
    UINT32 Length;
} EFI_USER_INFO_IDENTITY_POLICY;
```

Parameters

- **Type**
  Specifies either an operator or a data item. See `EFI_USER_INFO_IDENTITY_*` in “Related Definitions” below.

- **Length**
  The length of this block, in bytes, including this header.
**Description**

The user identity policy is an expression made up of operators and data items. If the expression evaluates to **TRUE**, then this user profile can be selected as the current profile. If the expression evaluates to **FALSE**, then this user profile cannot be selected as the current profile.

Data items are pushed onto an expression stack. Operators pop items off of the expression stack, perform an operator and push the results back.

**Note:** If there is no user identity policy set for a user profile, then **FALSE** is assumed.

**Access**

Exclusive:Yes

Modify:Only with user-enrollment permissions.

Visibility:Public.

**Related Definitions**

```c
#define EFI_USER_INFO_IDENTITY_FALSE 0x00
#define EFI_USER_INFO_IDENTITY_TRUE 0x01
#define EFI_USER_INFO_IDENTITY_CREDENTIAL_TYPE 0x02
#define EFI_USER_INFO_IDENTITY_CREDENTIAL_PROVIDER 0x03
#define EFI_USER_INFO_IDENTITY_NOT 0x10
#define EFI_USER_INFO_IDENTITY_AND 0x11
#define EFI_USER_INFO_IDENTITY_OR 0x12
```

<table>
<thead>
<tr>
<th>Type Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>EFI_USER_INFO_IDENTITY_FALSE</td>
<td>Push <strong>FALSE</strong> on to the expression stack.</td>
</tr>
<tr>
<td>EFI_USER_INFO_IDENTITY_TRUE</td>
<td>Push <strong>TRUE</strong> on to the expression stack.</td>
</tr>
<tr>
<td>EFI_USER_INFO_IDENTITY_CREDENTIAL_TYPE</td>
<td>If a credential provider with the specified class asserts the user’s identity, push <strong>TRUE</strong>. Otherwise push <strong>FALSE</strong>. The EFI_USER_INFO_IDENTITY_POLICY structure is followed immediately by a GUID.</td>
</tr>
<tr>
<td>EFI_USER_INFO_IDENTITY_CREDENTIAL_PROVIDER</td>
<td>If a credential provider with the specified provider identifier asserts the user’s identity, push <strong>TRUE</strong>. Otherwise, push <strong>FALSE</strong>. The EFI_USER_INFO_IDENTITY_POLICY structure is followed immediately by a GUID.</td>
</tr>
<tr>
<td>EFI_USER_INFO_IDENTITY_NOT</td>
<td>Pop a boolean off the stack. If <strong>TRUE</strong>, then push <strong>FALSE</strong>. If <strong>FALSE</strong>, then push <strong>TRUE</strong>.</td>
</tr>
<tr>
<td>EFI_USER_INFO_IDENTITY_AND</td>
<td>Pop two Booleans off the stack. If both are <strong>TRUE</strong>, then push <strong>TRUE</strong>. Otherwise push <strong>FALSE</strong>.</td>
</tr>
</tbody>
</table>
36.4.12 EFI_USER_INFO_NAME_RECORD

Summary
Provide the user’s name for the enrolled user.

Prototype
```c
#define EFI_USER_INFO_NAME_RECORD 0x01
typedef CHAR16 *EFI_USER_INFO_NAME;
```

Description
The user’s name is a NULL-terminated string.

Access
Exclusive:Yes
Visibility:Public.

36.4.13 EFI_USER_INFO_PKCS11_RECORD

Summary
Provides PKCS#11 credential information from a smart card.

Prototype
```c
#define EFI_USER_INFO_PKCS11_RECORD 0x0A
```

36.4.14 EFI_USER_INFO_RETRY_RECORD

Summary
Indicates how many attempts the user has to with a particular credential before the system prevents further attempts.

Prototype
```c
#define EFI_USER_INFO_RETRY_RECORD 0x0D
typedef UINT8 EFI_USER_INFO_RETRY;
```

Description
This record indicates the number of times the user may fail identification with all credential providers (EFI_USER_INFO.Credential is zero) or a particular credential provider (EFI_USER_INFO.Credential is non-zero).

Access:
Exclusive:No
Modify: Only with user-enrollment permissions.

Visibility: Public

### 36.4.15 EFI_USER_INFO_USAGE_DATE_RECORD

**Summary**

Provides the date and time when the user profile was selected.

**Prototype**

```c
#define EFI_USER_INFO_USAGE_DATE_RECORD 0x03
typedef EFI_TIME EFI_USER_INFO_USAGE_DATE;
```

**Description**

The optional record describing the date and time when the user profile was last selected. Type **EFI_TIME** is defined in **GetTime()** in this specification.

### 36.4.16 EFI_USER_INFO_USAGE_COUNT_RECORD

**Summary**

Provides the number of times that the user profile has been selected.

**Prototype**

```c
#define EFI_USER_INFO_USAGE_COUNT 0x04
typedef UINT64 EFI_USER_INFO_USAGE_COUNT;
```

**Description**

The optional record describing the number of times that the user profile was selected.

### 36.5 User Information Table

**Summary**

A collection of **EFI_USER_INFO** records, prefixed with this header.

**Prototype**

```c
typedef struct {
    UINT64 Size;
} EFI_USER_INFO_TABLE;
```

**Members**

- **Size**

  Total size of the user information table, in bytes.

**Description**

This header is followed by a series of records. Each record is prefixed by the **EFI_USER_INFO** structure. The total size of this header and all records is equal to **Size**.
37 - Secure Technologies

37.1 Hash Overview

For the purposes of this specification, a hash function takes a variable length input and generates a fixed length hash value. In general, hash functions are collision-resistant, which means that it is infeasible to find two distinct inputs which produce the same hash value. Hash functions are generally one-way which means that it is infeasible to find an input based on the output hash value.

This specification describes a protocol which allows a driver to produce a protocol which supports zero or more hash functions.

37.1.1 Hash References

The following references define the standard means of creating the hashes used in this specification:


For more information

- see “Links to UEFI-Related Documents” ([http://uefi.org/uefi](http://uefi.org/uefi)) under the heading “Archived FIPS publication”.
- see “Links to UEFI-Related Documents” ([http://uefi.org/uefi](http://uefi.org/uefi)) under the heading “MD5 Message-Digest Algorithm”. EFI Hash Protocols

**EFI_HASH_SERVICE_BINDING_PROTOCOL**

**Summary**

The EFI Hash Service Binding Protocol is used to locate hashing services support provided by a driver and create and destroy instances of the EFI Hash Protocol so that a multiple drivers can use the underlying hashing services.

The EFI Service Binding Protocol that is defined in Section 2.5.8 defines the generic Service Binding Protocol functions. This section discusses the details that are specific to the EFI Hash Protocol.

**GUID**

```c
#define EFI_HASH_SERVICE_BINDING_PROTOCOL_GUID \ 
{0x42881c98,0xa4f3,0x44b0,\ 
{0xa3,0x9d,0xdf,0xa1,0x86,0x67,0xd8,0xcd}}
```

**Description**

An application (or driver) that requires hashing services can use one of the protocol handler services, such as `BS->LocateHandleBuffer()`, to search for devices that publish an EFI Hash Service Binding Protocol.

After a successful call to the `EFI_HASH_SERVICE_BINDING_PROTOCOL.CreateChild()` function, the child EFI Hash Protocol driver instance is ready for use. The instance of `EFI_HASH_PROTOCOL` must be
obtained by performing `HandleProtocol()` against the handle returned by `CreateChild()`. Use of other methods, such as `LocateHandle()`, are not supported.

Once obtained, the driver may use the `EFI_HASH_PROTOCOL` instance for any number of non-overlapping hash operations. Overlapping hash operations require an additional call to `EFI_HASH_SERVICE_BINDING_PROTOCOL.CreateChild()` for a new instance.

Before a driver or application terminates execution, every successful call to the `EFI_HASH_SERVICE_BINDING_PROTOCOL.CreateChild()` function must be matched with a call to the `EFIHASH_SERVICE_BINDING_PROTOCOL.DestroyChild()` function.

**EFI_HASH_PROTOCOL**

**Summary**

This protocol describes standard hashing functions.

**GUID**

```c
#define EFI_HASH_PROTOCOL_GUID \
{0xc5184932,0xdba5,0x46db,\ 
{0xa5,0xba,0xcc,0x0b,0xda,0x9c,0x14,0x35}}
```

**Protocol Interface Structure**

```c
typedef _EFI_HASH_PROTOCOL {
    EFI_HASH_GET_HASH_SIZE GetHashSize;
    EFI_HASH_HASH Hash;
} EFI_HASH_PROTOCOL;
```

**Parameters**

- **GetHashSize**
  Return the size of a specific type of resulting hash.

- **Hash**
  Create a hash for the specified message.

**Description**

This protocol allows creating a hash of an arbitrary message digest using one or more hash algorithms. The `GetHashSize` returns the expected size of the hash for a particular algorithm and whether or not that algorithm is, in fact, supported. The `Hash` actually creates a hash using the specified algorithm.

**Related Definitions**

None.

**EFI_HASH_PROTOCOL.GetHashSize()**

**Summary**

Returns the size of the hash which results from a specific algorithm.
Prototype

```c
EFI_STATUS
EFIAPI
GetHashSize(
    IN CONST EFI_HASH_PROTOCOL *This,
    IN CONST EFI_GUID *HashAlgorithm,
    OUT UINTN *HashSize
);
```

Parameters

- `This` Points to this instance of `EFI_HASH_PROTOCOL`.
- `HashAlgorithm` Points to the `EFI_GUID` which identifies the algorithm to use. See Section 37.1.2.1.
- `HashSize` Holds the returned size of the algorithm's hash.

Description

This function returns the size of the hash which will be produced by the specified algorithm.

Related Definitions

None

Status Codes Returned

<table>
<thead>
<tr>
<th>Status Code</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>EFI_SUCCESS</td>
<td>Hash size returned successfully.</td>
</tr>
<tr>
<td>EFI_INVALID_PARAMETER</td>
<td><code>HashSize</code> is <code>NULL</code> or <code>HashAlgorithm</code> is <code>NULL</code>.</td>
</tr>
<tr>
<td>EFI_UNSUPPORTED</td>
<td>The algorithm specified by <code>HashAlgorithm</code> is not supported by this driver.</td>
</tr>
</tbody>
</table>

`EFI_HASH_PROTOCOL.Hash()`

Summary

Creates a hash for the specified message text.

Prototype

```c
EFI_STATUS
EFIAPI
Hash(
    IN CONST EFI_HASH_PROTOCOL *This,
    IN CONST EFI_GUID *HashAlgorithm,
    IN BOOLEAN Extend,
    IN CONST UINT8 *Message,
    IN UINT64 MessageSize,
    IN OUT EFI_HASH_OUTPUT *Hash
```
Parameters

This
Points to this instance of EFI_HASH_PROTOCOL.

HashAlgorithm
Points to the EFI_GUID which identifies the algorithm to use. See Section 37.1.2.1.

Extend
Specifies whether to create a new hash (FALSE) or extend the specified existing hash (TRUE).

Message
Points to the start of the message.

MessageSize
The size of Message, in bytes. Must be integer multiple of block size.

Hash
On input, if Extend is TRUE, then this parameter holds a pointer to a pointer to an array containing the hash to extend. If Extend is FALSE, then this parameter holds a pointer to a pointer to a caller-allocated array that will receive the result of the hash computation. On output (regardless of the value of Extend), the array will contain the result of the hash computation.

Description

This function creates the hash of the specified message text based on the specified algorithm HashAlgorithm and copies the result to the caller-provided buffer Hash. If Extend is TRUE, then the hash specified on input by Hash is extended. If Extend is FALSE, then the starting hash value will be that specified by the algorithm.

Note: For the all algorithms used with EFI_HASH_PROTOCOL, the following apply:

- The EFI_HASH_PROTOCOL.Hash() function does not perform padding of message data for these algorithms. Hence, MessageSize shall always be an integer multiple of the HashAlgorithm block size, and the final supplied Message in a sequence of invocations shall contain caller-provided padding. This will ensure that the final Hash output will be the correct hash of the provided message(s).
- The result of a Hash() call for one of these algorithms when the caller does not supply message data whose length is an integer multiple of the algorithm’s block size is a returned error.
- The EFI_HASH_OUTPUT options for these two algorithms shall be EFI_SHA1_HASH and EFI_SHA256_HASH, respectively.
- Callers using these algorithms may consult the aforementioned Secure Hash Standard for details on how to perform proper padding required by standard prior to final invocation.

Related Definitions

EFI_HASH_OUTPUT
Status Codes Returned

<table>
<thead>
<tr>
<th>Code</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>EFI_SUCCESS</td>
<td>Hash returned successfully.</td>
</tr>
<tr>
<td>EFI_INVALID_PARAMETER</td>
<td>Message or Hash, HashAlgorithm is NULL or MessageSize is 0. MessageSize is not an integer multiple of block size.</td>
</tr>
<tr>
<td>EFI_UNSUPPORTED</td>
<td>The algorithm specified by HashAlgorithm is not supported by this driver. Includes HashAlgorithm being passed as a null error.</td>
</tr>
<tr>
<td>EFI_UNSUPPORTED</td>
<td>Extend is TRUE and the algorithm doesn’t support extending the hash.</td>
</tr>
</tbody>
</table>

37.1.2 Other Code Definitions

**EFI_SHA1_HASH, EFI_SHA224_HASH, EFI_SHA256_HASH, EFI_SHA384_HASH, EFI_SHA512HASH, EFI_MD5_HASH**

Summary
Data structure which holds the result of the hash.

Prototype

```c
typedef UINT8 EFI_MD5_HASH[16];
typedef UINT8 EFI_SHA1_HASH[20];
typedef UINT8 EFI_SHA224_HASH[28];
typedef UINT8 EFI_SHA256_HASH[32];
typedef UINT8 EFI_SHA384_HASH[48];
typedef UINT8 EFI_SHA512_HASH[64];
typedef union _EFI_HASH_OUTPUT {
    EFI_MD5_HASH  *Md5Hash;
    EFI_SHA1_HASH *Sha1Hash;
    EFI_SHA224_HASH *Sha224Hash;
    EFI_SHA256_HASH *Sha256Hash;
    EFI_SHA384_HASH *Sha384Hash;
    EFI_SHA512_HASH *Sha512Hash;
} EFI_HASH_OUTPUT;
```

Description
These prototypes describe the expected hash output values from the Hash function of the EFI_HASH_PROTOCOL.

Related Definitions
None

37.1.2.1 EFI Hash Algorithms
The following table gives the EFI_GUID for standard hash algorithms and the corresponding ASN.1 OID (Object Identifier):
**Note:** Use of the following algorithms with EFI_HASH_PROTOCOL is deprecated.

- EFI_HASH_ALGORITHM_SHA1_GUID
- EFI_HASH_ALGORITHM_SHA224_GUID
- EFI_HASH_ALGORITHM_SHA256_GUID
- EFI_HASH_ALGORITHM_SHA384_GUID
- EFI_HASH_ALGORITHM_SHA512_GUID
- EFI_HASH_ALGORITHM_MD5_GUID

### Table 37-1 EFI Hash Algorithms

<table>
<thead>
<tr>
<th>Algorithm</th>
<th>EFW_GUID</th>
<th>OID</th>
</tr>
</thead>
<tbody>
<tr>
<td>SHA-1 (No padding done by implementation)</td>
<td>#define EFI_HASH_ALGORITHM_SHA1_NOPAD_GUID {0x24c5dc2f, 0x53e2, 0x40ca, {0x9e, 0xd6, 0xa5, 0xd9, 0xa4, 0x9f, 0x46, 0x3b}}</td>
<td>id-sha1 OBJECT IDENTIFIER ::= { iso(1) identified-organization(3) oiw(14) secsig(3) algorithms(2) 26 }</td>
</tr>
<tr>
<td>SHA-256 (No padding done by implementation)</td>
<td>#define EFI_HASH_ALGORITHM_SHA256_NOPAD_GUID {0x8628752a, 0x6cb7, 0x4814, {0x96, 0xfc, 0x24, 0xa8, 0x15, 0xac, 0x22, 0x26}}</td>
<td>id-sha256 OBJECT IDENTIFIER ::= { joint-iso-itu-t (2) country (16) us (840) organization (1) gov (101) csor (3) nistalgorithm (4) hashalgs (2) 1 }</td>
</tr>
</tbody>
</table>

**Note:** For the EFI_HASH_ALGORITHM_SHA1_NOPAD_GUID and the EFI_HASH_ALGORITHM_SHA256_NOPAD_GUID, the following apply:

- The EFI_HASH_PROTOCOL.Hash() function does not perform padding of message data for these algorithms. Hence, MessageSize shall always be an integer multiple of the HashAlgorithm block size, and the final supplied Message in a sequence of invocations shall contain caller-provided padding. This will ensure that the final Hash output will be the correct hash of the provided message(s).
- The result of a Hash() call for one of these algorithms when the caller does not supply message data whose length is an integer multiple of the algorithm’s block size is undefined.
- The EFI_HASH_OUTPUT options for these two algorithms shall be EFI_SHA1_HASH and EFI_SHA256_HASH, respectively.
- Callers using these algorithms may consult the aforementioned Secure Hash Standard for details on how to perform proper padding.
37.2 Hash2 Protocols

37.2.1 EFI Hash2 Service Binding Protocol

EFI_HASH2_SERVICE_BINDING_PROTOCOL

Summary
The EFI Hash2 Service Binding Protocol is used to locate EFI_HASH2_PROTOCOL hashing services support provided by a driver and create and destroy instances of the EFI_HASH2_PROTOCOL Protocol so that a multiple drivers can use the underlying hashing services.

The EFI Service Binding Protocol that is defined in Section 2.5.8 defines the generic Service Binding Protocol functions. This section discusses the details that are specific to the EFI Hash Protocol.

GUID

```
#define EFI_HASH2_SERVICE_BINDING_PROTOCOL_GUID
{0xda836f8d, 0x217f, 0x4ca0, 0x99, 0xc2, 0x1c, 0xa4, 0xe1, 0x60, 0x77, 0xea}
```

Description
An application (or driver) that requires hashing services can use one of the protocol handler services, such as BS->LocateHandleBuffer(), to search for devices that publish an EFI_HASH2_SERVICE_BINDING_PROTOCOL.

After a successful call to the EFI_HASH2_SERVICE_BINDING_PROTOCOL member CreateChild() function, the child instance of EFI_HASH2_PROTOCOL Protocol driver instance is ready for use. The instance of EFI_HASH2_PROTOCOL must be obtained by performing HandleProtocol() against the handle returned by CreateChild(). Use of other methods, such as LocateHandle() is not supported.

Once obtained, the driver may use the EFI_HASH2_PROTOCOL instance for any number of non-overlapping hash operations. Overlapping hash operations require an additional call to EFI_HASH_SERVICE_BINDING_PROTOCOL.CreateChild() for a new instance.

Before a driver or application using the instance terminates execution, every successful call to the EFI_HASH_SERVICE_BINDING_PROTOCOL.CreateChild() function must be matched with a call to the EFI_HASH_SERVICE_BINDING_PROTOCOL.DestroyChild() function.

37.2.2 EFI Hash2 Protocol

EFI_HASH2_PROTOCOL

Summary
This protocol describes hashing functions for which the algorithm-required message padding and finalization are performed by the supporting driver. In previous versions of the specification, the algorithms supported by EFI_HASH2_PROTOCOL were also available for use with EFI_HASH_PROTOCOL but this usage has been deprecated.
GUID

```c
#define EFI_HASH2_PROTOCOL_GUID \
{ 0x55b1d734, 0xc5e1, 0x49db, 0x96, 0x47, 0xb1, \n 0xfb, 0xe, 0x30, 0x5b}
```

Protocol Interface Structure

```c
typedef _EFI_HASH2_PROTOCOL {
  EFI_HASH2_GET_HASH_SIZE GetHashSize;
  EFI_HASH2_HASH Hash;
  EFI_HASH2_HASH_INIT HashInit;
  EFI_HASH2_HASH_UPDATE HashUpdate;
  EFI_HASH2_HASH_FINAL HashFinal;
} EFI_HASH2_PROTOCOL;
```

Parameters

- **GetHashSize**
  Return the result size of a specific type of resulting hash.

- **Hash**
  Create a final non-extendable hash for a single message block in a single call.

- **HashInit**
  Initializes an extendable multi-part hash calculation

- **HashUpdate**
  Continues a hash in progress by supplying the first or next sequential portion of the message text

- **HashFinal**
  Finalizes a hash in progress by padding as required by algorithm and returning the hash output.

Description

This protocol allows creating a hash of an arbitrary message digest using one or more hash algorithms. The **GetHashSize()** function returns the expected size of the hash for a supported algorithm and an error if that algorithm is not supported. The **Hash()** function creates a final, non-extendable, hash of a single message block using the specified algorithm. The three functions **HashInit()**, **HashUpdate()**, **HashFinal()**, generates the hash of a multi-part message, with input composed of one or more message pieces.

For a specific handle representing an instance of **EFI_HASH2_PROTOCOL**, if **Hash()** is called after a call to **HashInit()** and prior to the matching call to **HashFinal()**, the multi-part hash started by **HashInit()** will be canceled and calls to **HashUpdate()** or **HashFinal()** will return an error status unless proceeded by a new call to **HashInit()**.

**Note:** Algorithms **EFI_HASH_ALGORITHM_SHA1_NOPAD** and **EFI_HASH_ALGORITHM_SHA256_NOPAD_GUID** are not compatible with **EFI_HASH2_PROTOCOL** and will return **EFI_UNSUPPORTED** if used with any **EFI_HASH2_PROTOCOL** function.

Related Definitions

None
Note: The following hash function invocations will produce identical hash results for all supported EFI_HASH2_PROTOCOL algorithms. The data in quotes is the message.

Table 37-2 Identical hash results

<table>
<thead>
<tr>
<th>Hash(&quot;ABCDEF&quot;)</th>
<th>HashInit()</th>
<th>HashInit()</th>
</tr>
</thead>
<tbody>
<tr>
<td>HashUpdate(&quot;ABCDEF&quot;)</td>
<td>HashUpdate(&quot;ABC&quot;)</td>
<td>HashUpdate(&quot;DE&quot;)</td>
</tr>
<tr>
<td>HashFinal()</td>
<td>HashUpdate(&quot;F&quot;)</td>
<td>HashFinal()</td>
</tr>
</tbody>
</table>
EFI_HASH2_PROTOCOL.GetHashSize()

Summary
Returns the size of the hash which results from a specific algorithm.

Prototype

```c
EFI_STATUS
EFIAPI
GetHashSize(
    IN CONST EFI_HASH2_PROTOCOL *This,
    IN CONST EFI_GUID *HashAlgorithm,
    OUT UINTN *HashSize
);
```

Parameters

- **This**
  Points to this instance of **EFI_HASH2_PROTOCOL**.
- **HashAlgorithm**
  Points to the **EFI_GUID** which identifies the algorithm to use. See Section 37.2.3
- **HashSize**
  Holds the returned size of the algorithm's hash.

Description
This function returns the size of the hash result buffer which will be produced by the specified algorithm.

Related Definitions
None
Status Codes Returned

<table>
<thead>
<tr>
<th>Status Code</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>EFI_SUCCESS</td>
<td>Hash size returned successfully.</td>
</tr>
<tr>
<td>EFI_INVALID_PARAMETER</td>
<td>This or HashSize is NULL</td>
</tr>
<tr>
<td>EFI_UNSUPPORTED</td>
<td>The algorithm specified by HashAlgorithm is not supported by this driver or, HashAlgorithm is null.</td>
</tr>
</tbody>
</table>

**EFI_HASH2_PROTOCOL.Hash()**

**Summary**

Creates a hash for a single message text. The hash is not extendable. The output is final with any algorithm-required padding added by the function.

**Prototype**

```c
EFI_STATUS
EFILEAPI
Hash(
    IN CONST EFI_HASH2_PROTOCOL *This,
    IN CONST EFI_GUID *HashAlgorithm,
    IN CONST UINT8 *Message,
    IN UINTN MessageSize,
    IN OUT EFI_HASH2_OUTPUT *Hash
);```

**Parameters**

- **This**
  Points to this instance of **EFI_HASH2_PROTOCOL**.
- **HashAlgorithm**
  Points to the **EFI_GUID** which identifies the algorithm to use. See Table 37-3.
- **Message**
  Points to the start of the message.
- **MessageSize**
  The size of Message, in bytes.
Hash

On input, points to a caller-allocated buffer of the size returned by \texttt{GetHashSize()} for the specified \texttt{HashAlgorithm}. On output, the buffer holds the resulting hash computed from the message.

Description

This function creates the hash of specified single block message text based on the specified algorithm \texttt{HashAlgorithm} and copies the result to the caller-provided buffer \texttt{Hash}. The resulting hash cannot be extended. All padding required by \texttt{HashAlgorithm} is added by the implementation.

Related Definitions

\texttt{EFI_HASH2_OUTPUT}

Status Codes Returned

<table>
<thead>
<tr>
<th>Status Code</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>\texttt{EFI_SUCCESS}</td>
<td>\texttt{Hash} returned successfully.</td>
</tr>
<tr>
<td>\texttt{EFI_INVALID_PARAMETER}</td>
<td>\texttt{This}, or \texttt{Hash} is NULL.</td>
</tr>
<tr>
<td>\texttt{EFI_UNSUPPORTED}</td>
<td>The algorithm specified by \texttt{HashAlgorithm} is not supported by this driver or \texttt{HashAlgorithm} is NULL.</td>
</tr>
<tr>
<td>\texttt{EFI_OUT_OF_RESOURCES}</td>
<td>Some resource required by the function is not available or \texttt{MessageSize} is greater than platform maximum.</td>
</tr>
</tbody>
</table>

\texttt{EFI_HASH2_PROTOCOL.HashInit()}

Summary

This function must be called to initialize a digest calculation to be subsequently performed using the \texttt{EFI_HASH2_PROTOCOL} functions \texttt{HashUpdate()} and \texttt{HashFinal()}. 

Prototype

\begin{verbatim}
EFI_STATUS
EFIAPI
HashInit(
    IN CONST EFI_HASH2_PROTOCOL *This,
    IN CONST EFI_GUID *HashAlgorithm,
); 
\end{verbatim}

Parameters

\begin{verbatim}
This
HashAlgorithm
\end{verbatim}

Points to instance of \texttt{EFI_HASH2_PROTOCOL}. Points to the \texttt{EFI_GUID} which identifies the algorithm to use. See \texttt{Table 37-3}.

Description

This function
Related Definitions

Status Codes Returned

<table>
<thead>
<tr>
<th>Status Code</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>EFI_SUCCESS</td>
<td>Initialized successfully.</td>
</tr>
<tr>
<td>EFI_INVALID_PARAMETER</td>
<td>This is NULL.</td>
</tr>
<tr>
<td>EFI_UNSUPPORTED</td>
<td>The algorithm specified by HashAlgorithm is not supported by this function or HashAlgorithm is NULL.</td>
</tr>
<tr>
<td>EFI_OUT_OF_RESOURCES</td>
<td>Process failed due to lack of required resource.</td>
</tr>
<tr>
<td>EFI_ALREADY_STARTED</td>
<td>This function is called when the operation in progress is still in processing Hash(), or HashInit() is already called before and not terminated by HashFinal() yet on the same instance.</td>
</tr>
</tbody>
</table>

**EFI_HASH2_PROTOCOL.HashUpdate()**

**Summary**
Updates the hash of a computation in progress by adding a message text.

**Prototype**

```c
EFI_STATUS
EFI_API
HashUpdate(
   IN CONST EFI_HASH2_PROTOCOL *This,
   IN CONST UINT8 *Message,
   IN UINTN MessageSize
);
```

**Parameters**

- **This** Points to instance of EFI_HASH2_PROTOCOL.
- **Message** Points to the start of the message.
- **MessageSize** The size of Message, in bytes.

**Description**
This function extends the hash of ongoing hash operation with the supplied message text. This function should be called one or more times with portions of the total message text to be hashed. A zero-length message input will return EFI_SUCCESS and has no impacts on the ongoing hash instance.
Related Definitions

Status Codes Returned

<table>
<thead>
<tr>
<th>Code</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>EFI_SUCCESS</td>
<td>Digest in progress updated successfully.</td>
</tr>
<tr>
<td>EFI_INVALID_PARAMETER</td>
<td>This or Hash is NULL.</td>
</tr>
<tr>
<td>EFI_OUT_OF_RESOURCES</td>
<td>Some resource required by the function is not available or MessageSize is greater than platform maximum.</td>
</tr>
<tr>
<td>EFI_NOT_READY</td>
<td>This call was not preceded by a valid call to HashInit(), or the operation in progress was terminated by a call to Hash() or HashFinal() on the same instance.</td>
</tr>
</tbody>
</table>

**EFI_HASH2_PROTOCOL.HashFinal()**

**Summary**
Finalizes a hash operation in progress and returns calculation result. The output is final with any necessary padding added by the function. The hash may not be further updated or extended after HashFinal().

**Prototype**
```c
EFI_STATUS
EFIAPI
HashFinal(
    IN CONST EFI_HASH2_PROTOCOL *This,
    IN OUT EFI_HASH2_OUTPUT *Hash
);
```

**Parameters**
- **This** Points to instance of EFI_HASH2_PROTOCOL.
- **Hash** On input, points to a caller-allocated buffer of the size returned by GetHashSize() for the specified HashAlgorithm specified in preceding HashInit(). On output, the buffer holds the resulting hash computed from the message.

**Description**
This function finalizes the hash of a hash operation in progress. The resulting final hash cannot be extended.
Related Definitions

**EFI_HASH2_OUTPUT**

Status Codes Returned

<table>
<thead>
<tr>
<th>Status Code</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>EFI_SUCCESS</td>
<td>Hash returned successfully.</td>
</tr>
<tr>
<td>EFI_INVALID_PARAMETER</td>
<td>This or Hash is NULL.</td>
</tr>
<tr>
<td>EFI_NOT_READY</td>
<td>This call was not preceded by a valid call to <strong>HashInit()</strong> and at least one call to <strong>HashUpdate()</strong>, or the operation in progress was canceled by a call to <strong>Hash()</strong> on the same instance.</td>
</tr>
</tbody>
</table>

Table 37-3 Algorithms that may be used with EFI_HASH2_PROTOCOL

<table>
<thead>
<tr>
<th>Algorithm</th>
<th>EFI_GUID</th>
<th>OID</th>
</tr>
</thead>
<tbody>
<tr>
<td>SHA-1</td>
<td>#define</td>
<td></td>
</tr>
<tr>
<td></td>
<td>EFI_HASH_ALGORITHM_SHA1_GUID {0x2ae9d80f, 0x3fb2, 0x4095, { 0xb7, 0xb1, 0xe9, 0x31, 0x57, 0xb9, 0x46, 0xb6}}</td>
<td>id-sha1 OBJECT IDENTIFIER ::= { iso(1) identified-organization(3) oiw(14) secsig(3) algorithms(2) 26 }</td>
</tr>
<tr>
<td>SHA-224</td>
<td>#define</td>
<td></td>
</tr>
<tr>
<td></td>
<td>EFI_HASH_ALGORITHM_SHA224_GUID { 0x8df01a06, 0x9bd5, 0x4bf7, { 0xb0, 0x21, 0xdb, 0x4f, 0xd9, 0xcc, 0xf4, 0x5b } }</td>
<td></td>
</tr>
<tr>
<td>SHA-256</td>
<td>#define</td>
<td></td>
</tr>
<tr>
<td></td>
<td>EFI_HASH_ALGORITHM_SHA256_GUID { 0x51aa59de, 0xfdf2, 0x4ea3, { 0xbc, 0x63, 0x87, 0x5f, 0xb7, 0x84, 0xe2e, 0xe9 } }</td>
<td>id-sha256 OBJECT IDENTIFIER ::= { joint-iso-itu-t (2) country (16) us (840) organization (1) gov (101) csor (3) nistalgorithm (4) hashalgs (2) 1}</td>
</tr>
<tr>
<td>SHA-384</td>
<td>#define</td>
<td></td>
</tr>
<tr>
<td></td>
<td>EFI_HASH_ALGORITHM_SHA384_GUID { 0xea96432, 0xde33, 0x4dd2, { 0xae, 0xe6, 0x32, 0x8c, 0x33, 0xdf, 0x777, 0x7a } }</td>
<td>id-sha384 OBJECT IDENTIFIER ::= { joint-iso-itu-t (2) country (16) us (840) organization (1) gov (101) csor (3) nistalgorithm (4) hashalgs (2) 2}</td>
</tr>
</tbody>
</table>
37.2.3 Other Code Definitions

**EFI_HASH2_OUTPUT**

**Summary**

Data structure which holds the result of the hash operation from `EFI_HASH2_PROTOCOL` hash operations.

**Prototype**

```c
typedef UINT8 EFI_MD5_HASH2[16];
typedef UINT8 EFI_SHA1_HASH2[20];
typedef UINT8 EFI_SHA224_HASH2[28];
typedef UINT8 EFI_SHA256_HASH2[32];
typedef UINT8 EFI_SHA384_HASH2[48];
typedef UINT8 EFI_SHA512_HASH2[64];
typedef union _EFI_HASH2_OUTPUT {
    EFI_MD5_HASH2     Md5Hash;
    EFI_SHA1_HASH2    Sha1Hash;
    EFI_SHA224_HASH2  Sha224Hash;
    EFI_SHA256_HASH2  Sha256Hash;
    EFI_SHA384_HASH2  Sha384Hash;
    EFI_SHA512_HASH2  Sha512Hash;
} EFI_HASH2_OUTPUT;
```

**Note:** SHA-1 and MD5 are included for backwards compatibility. New driver implementations are encouraged to consider stronger algorithms.
Description
These prototypes describe the expected hash output values from the hashing functions of the EFI_HASH2_PROTOCOL.

Related Definitions
None

37.3 Key Management Service

EFI_KEY_MANAGEMENT_SERVICE_PROTOCOL

Summary
The Key Management Service (KMS) protocol provides services to generate, store, retrieve, and manage cryptographic keys. The intention is to specify a simple generic protocol that could be used for many implementations.

The management keys have a simple construct – they consist of key identifier and key data, both of variable size.

A driver implementing the protocol may need to provide basic key service that consists of a key store and cryptographic key generation capability. It may connect to an external key server over the network, or to a Hardware Security Module (HSM) attached to the system it runs on, or anything else that is capable of providing the key management service.

Authentication and access control is not addressed by this protocol. It is assumed it is addressed at the system level and done by the driver implementing the protocol, if applicable to the implementation.

GUID

```
#define EFI_KMS_PROTOCOL_GUID \
{0xEC3A978D,0x7C4E, 0x48FA, \
{0x9A,0xBE,0x6A,0xD9,0x1C,0xC8,0xF8,0x11}}
```

Protocol Interface Structure

```
#define EFI_KMS_DATA_TYPE_NONE  0
#define EFI_KMS_DATA_TYPE_BINARY 1
#define EFI_KMS_DATA_TYPE_ASCII  2
#define EFI_KMS_DATA_TYPE_UNICODE 4
#define EFI_KMS_DATA_TYPE_UTF8  8
```

Where appropriate, EFI_KMS_DATA_TYPE values may be combined using a bitwise ‘OR’ operation to indicate support for multiple data types.
typedef struct _EFI_KMS_SERVICE_PROTOCOL {
    EFI_KMS_GET_SERVICE_STATUS GetServiceStatus;
    EFI_KMS_REGISTER_CLIENT RegisterClient;
    EFI_KMS_CREATE_KEY CreateKey;
    EFI_KMS_GET_KEY GetKey;
    EFI_KMS_ADD_KEY AddKey;
    EFI_KMS_DELETE_KEY DeleteKey;
    EFI_KMS_GET_KEY_ATTRIBUTES GetKeyAttributes;
    EFI_KMS_ADD_KEY_ATTRIBUTES AddKeyAttributes;
    EFI_KMS_DELETE_KEY_ATTRIBUTES DeleteKeyAttributes;
    EFI_KMS_GET_KEY_BY_ATTRIBUTES GetKeyByAttributes;
    UINT32 ProtocolVersion;
    EFI_GUID ServiceId;
    CHAR16 *ServiceName;
    UINT32 ServiceVersion;
    BOOLEAN ServiceAvailable;
    BOOLEAN ClientIdSupported;
    BOOLEAN ClientIdRequired;
    UINT16 ClientIdMaxSize;
    UINT8 ClientNameStringTypes;
    BOOLEAN ClientNameRequired;
    UINT16 ClientNameMaxCount;
    BOOLEAN ClientDataSupported;
    UINTN ClientDataMaxSize;
    BOOLEAN KeyIdVariableLenSupported;
    UINTN KeyIdMaxSize;
    UINTN KeyFormatsCount;
    EFI_GUID *KeyFormats;
    BOOLEAN KeyAttributesSupported;
    UINT8 KeyAttributeIdStringTypes;
    UINT16 KeyAttributeIdMaxCount;
    UINTN KeyAttributesCount;
    EFI_KMS_KEY_ATTRIBUTE *KeyAttributes;
} EFI_KMS_PROTOCOL;

Parameters

GetServiceStatus Get the current status of the key management service. If the implementation has not yet connected to the KMS, then a call to this function will initiate a connection. This is the only function that is valid for use prior to the service being marked available.

RegisterClient Register a specific client with the KMS.
CreateKey Request the generation of a new key and retrieve it.
GetKey Retrieve an existing key.
AddKey  Add a local key to the KMS database. If there is an existing key with this key identifier in the KMS database, it will be replaced with the new key.

DeleteKey Delete an existing key from the KMS database.

AddKeyAttributes Add attributes to an existing key in the KMS database.

GetKeyAttributes Get attributes for an existing key in the KMS database.

DeleteKeyAttributes Delete attributes for an existing key in the KMS database.

GetKeyByAttributes Get existing key(s) with the specified attributes.

ProtocolVersion The version of this EFI_KMS_PROTOCOL structure. This must be set to 0x00020040 for the initial version of this protocol.

ServiceId Optional GUID used to identify a specific KMS. This GUID may be supplied by the provider, by the implementation, or may be null. If it is null, then the ServiceName must not be null.

ServiceName Optional pointer to a unicode string which may be used to identify the KMS or provide other information about the supplier.

ServiceVersion Optional 32-bit value which may be used to indicate the version of the KMS provided by the supplier.

ServiceAvailable TRUE if and only if the service is active and available for use. To avoid unnecessary delays in POST, this protocol may be installed without connecting to the service. In this case, the first call to the GetServiceStatus() function will cause the implementation to connect to the supported service and mark it as available. The capabilities of this service as defined in the remainder of this protocol are not guaranteed to be valid until the service has been marked available. FALSE otherwise.

ClientIdSupported TRUE if and only if the service supports client identifiers. Client identifiers may be used for auditing, access control or any other purpose specific to the implementation. FALSE otherwise.

ClientIdRequired TRUE if and only if the service requires a client identifier in order to process key requests. FALSE otherwise.

ClientIdMaxSize The maximum size in bytes for the client identifier.

ClientNameStringTypes The client name string type(s) supported by the KMS service. If client names are not supported, this field will be set to EFI_KMS_DATA_TYPE_NONE. Otherwise, it will be set to the inclusive 'OR' of all client name formats supported. Client names may be used for auditing, access control or any other purpose specific to the implementation.

ClientNameRequired TRUE if and only if the KMS service requires a client name to be supplied to the service. FALSE otherwise.
**ClientNameMaxCount**
The maximum number of characters allowed for the client name.

**ClientDataSupported**

TRUE if and only if the service supports arbitrary client data requests. The use of client data requires the caller to have specific knowledge of the individual KMS service and should be used only if absolutely necessary.
FALSE otherwise.

**ClientDataMaxSize**
The maximum size in bytes for the client data. If the maximum data size is not specified by the KMS or it is not known, then this field must be filled with all ones.

**KeyIdVariableLenSupported**

TRUE if variable length key identifiers are supported.
FALSE if a fixed length key identifier is supported.

**KeyIdMaxLen**
If **KeyIdVariableLenSupported** is TRUE, this is the maximum supported key identifier length in bytes. Otherwise this is the fixed length of key identifier supported. Key ids shorter than the fixed length will be padded on the right with blanks.

**KeyFormatsCount**
The number of key format/size GUIDs returned in the **KeyFormats** field.

**KeyFormats**
A pointer to an array of **EFI_GUID** values which specify key formats/sizes supported by this KMS. Each format/size pair will be specified by a separate **EFI_GUID**. At least one key format/size must be supported. All formats/sizes with the same hashing algorithm must be contiguous in the array, and for each hashing algorithm, the key sizes must be in ascending order. See “Related Definitions” for GUIDs which identify supported key formats/sizes.

‘This list of GUIDs supported by the KMS is not required to be exhaustive, and the KMS may provide support for additional key formats/sizes. Users may request key information using an arbitrary GUID, but any GUID not recognized by the implementation or not supported by the KMS will return an error code of **EFI_UNSUPPORTED**.

**KeyAttributesSupported**

TRUE if key attributes are supported.
FALSE if key attributes are not supported.

**KeyAttributeIdStringTypes**
The key attribute identifier string type(s) supported by the KMS service. If key attributes are not supported, this field will be set to **EFI_KMS_DATA_TYPE_NONE**. Otherwise, it will be set to the inclusive ‘OR’ of all key attribute identifier string types
supported. **EFI_KMS_DATA_TYPE_BINARY** is not valid for this field.

**KeyAttributeIdMaxCount**
- The maximum number of characters allowed for the client name.

**KeyAttributesCount**
- The number of predefined **KeyAttributes** structures returned in the **KeyAttributes** parameter. If the KMS does not support predefined key attributes, or if it does not provide a method to obtain predefined key attributes data, then this field must be zero.

**KeyAttributes**
- A pointer to an array of **KeyAttributes** structures which contains the predefined attributes supported by this KMS. Each structure must contain a valid key attribute identifier and should provide any other information as appropriate for the attribute, including a default value if one exists. This variable must be set to **NULL** if the **KeyAttributesCount** variable is zero. It must point to a valid buffer if the **KeyAttributesCount** variable is non-zero.

This list of predefined attributes is not required to be exhaustive, and the KMS may provide additional predefined attributes not enumerated in this list. The implementation does not distinguish between predefined and used defined attributes, and therefore, predefined attributes not enumerated will still be processed to the KMS.

### Related Definitions

Functions defined for this protocol typically require the caller to provide information about the client, the keys to be processed, and/or attributes of the keys to be processed. Four structures, **EFI_KMS_CLIENTINFO**, **EFI_KMS_KEY_DESCRIPTOR**, **EFI_KMS_DYNAMIC_ATTRIBUTE**, and **EFI_KMS_KEY_ATTRIBUTE** define the information to be passed to these functions.

```c
typedef struct {
    UINT16 ClientIdSize;
    VOID* clientId;
    UINT8 ClientNameType;
    UINT8 ClientNameCount;
    VOID* ClientName;
} EFI_KMS_CLIENTINFO;
```

- **ClientIdSize**: The size in bytes for the client identifier.
- **ClientId**: Pointer to a valid client identifier.
- **ClientNameType**: The client name string type used by this client. The string type set here must be one of the string types reported in the **ClientNameStringTypes** field of the KMS protocol. If the KMS does not support client names, this field should be set to **EFI_KMS_DATA_TYPE_NONE**.
ClientNameCount

The size in characters for the client name. This field will be ignored if ClientNameStringType is set to **EFI_KMS_DATA_TYPE_NONE.** Otherwise, it must contain number of characters contained in the ClientName field.

ClientName

Pointer to a client name. This field will be ignored if ClientNameStringType is set to **EFI_KMS_DATA_TYPE_NONE.** Otherwise, it must point to a valid string of the specified type.

The key formats recognized by the KMS protocol are defined by an **EFI_GUID** which specifies a (key-algorithm, key-size) pair. The names of these GUIDs are in the format **EFI_KMS_KEY_(key-algorithm)_(_key-size)_GUID**, where the key-size is expressed in bits. The key formats recognized fall into three categories, generic (no algorithm), hash algorithms, and encrypted algorithms.

**Generic Key Data:**

The following GUIDs define formats that contain generic key data of a specific size in bits, but which is not associated with any specific key algorithm(s).

```c
#define EFI_KMS_FORMAT_GENERIC_128_GUID \
{0xec8a3d69,0x6ddf,0x4108,\ 
 {0x94,0x76,0x73,0x37,0xfc,0x52,0x21,0x36}}

#define EFI_KMS_FORMAT_GENERIC_160_GUID \
{0xa3b3e6f8,0xefca,0x4bc1,\ 
 {0x88,0xfb,0xcb,0x87,0x33,0x9b,0x25,0x79}}

#define EFI_KMS_FORMAT_GENERIC_256_GUID \
{0x70f64793,0xc323,0x4261,\ 
 {0xac,0x2c,0xd8,0x76,0xf2,0x7c,0x53,0x45}}

#define EFI_KMS_FORMAT_GENERIC_512_GUID \
{0x978fe043,0xd7af,0x422e,\ 
 {0x8a,0x92,0x2b,0x48,0xe4,0x63,0xbd,0xe6}}

#define EFI_KMS_FORMAT_GENERIC_1024_GUID \
{0x978fe043,0xdd7b,0x422e,\ 
 {0xb0,0x9c,0xe5,0xe4,0x63,0xbd,0xe6}}

#define EFI_KMS_FORMAT_GENERIC_2048_GUID \
{0x978fe043,0xdd7b,0x422e,\ 
 {0xb0,0x9c,0xe5,0xe4,0x63,0xbd,0xe6}}

#define EFI_KMS_FORMAT_GENERIC_3072_GUID \
{0x978fe043,0xdd7b,0x422e,\ 
 {0xb0,0x9c,0xe5,0xe4,0x63,0xbd,0xe6}}

#define EFI_KMS_FORMAT_GENERIC_DYNAMIC_GUID \
```


The **EFI_KMS_FORMAT_GENERIC_DYNAMIC_GUID** is defined for the key data with a size not defined by a certain key format GUID. The key value specified by this GUID is in format of structure **EFI_KMS_FORMAT_GENERIC_DYNAMIC**.

```c
typedef struct {
    UINT32  KeySize;
    UINT8    KeyData[1];
} EFI_KMS_FORMAT_GENERIC_DYNAMIC;
```

**KeySize** Length in bytes of the **KeyData**.

**KeyData** The data of the key.

**Hash Algorithm Key Data:**

These GUIDS define key data formats that contain data generated by basic hash algorithms with no cryptographic properties.

```c
#define EFI_KMS_FORMAT_MD2_128_GUID \
    {0x78be11c4,0xee44,0x4a22,\ 
    {0x9f,0x05,0x03,0x85,0x2e,0xc5,0xc9,0x78}}

#define EFI_KMS_FORMAT_MDC2_128_GUID \
    {0xf7ad60f8,0xefa8,0x44a3,\ 
    {0x91,0x13,0x23,0x1f,0x39,0x9e,0xb4,0xc7}}

#define EFI_KMS_FORMAT_MD4_128_GUID \
    {0xd1c17aa1,0xcac5,0x400f,0xbe,\ 
    {0x17,0xe2,0xa2,0xae,0x06,0x67,0x7c}}

#define EFI_KMS_FORMAT_MDC4_128_GUID \
    {0x3fa4f847,0xd8eb,0x4df4,\ 
    {0xbd,0x49,0x10,0x3a,0x0a,0x84,0x7b,0xbc}}

#define EFI_KMS_FORMAT_MD5_128_GUID \
    {0xdcbbc3662,0x9cda,0x4b52,\ 
    {0xa0,0x4c,0x82,0xeb,0x1d,0x23,0x48,0xc7}}

#define EFI_KMS_FORMAT_MD5SHA_128_GUID \
    {0x1c178237,0x6897,0x459e,\ 
    {0x9d,0x36,0x67,0xce,0x8e,0xf9,0x4f,0x76}}

#define EFI_KMS_FORMAT_SHA1_160_GUID \
    {0x453c5e5a,0x482d,0x43f0,\ 
    {0x87,0xc9,0x59,0x41,0xf3,0xa3,0x8a,0xc2}}
```
Encryption Algorithm Key Data:

These GUIDs define key data formats that contain data generated by cryptographic key algorithms. There may or may not be a separate data hashing algorithm associated with the key algorithm.

The encryption algorithms defined above have the following properties:
Table 37-4 Encryption algorithm properties.

<table>
<thead>
<tr>
<th>EFI_KMS_FORMAT</th>
<th>Encryption Description</th>
<th>Key Data Size</th>
<th>Hash Function</th>
</tr>
</thead>
<tbody>
<tr>
<td>AESXTS_128</td>
<td>Symmetric encryption using XTS-AES 128 bit keys</td>
<td>Key data is a concatenation of two fields of equal size for a total size of 256 bits</td>
<td>N/A</td>
</tr>
<tr>
<td>AESXTS_256</td>
<td>Symmetric encryption using block cipher XTS-AES 256 bit keys</td>
<td>Key data is a concatenation of two fields of equal size for a total size of 512 bits</td>
<td>N/A</td>
</tr>
<tr>
<td>AESCBC_128</td>
<td>Symmetric encryption using block cipher AES-CBC 128 bit keys</td>
<td>128 bits</td>
<td>N/A</td>
</tr>
<tr>
<td>AESCBC_256</td>
<td>Symmetric encryption using block cipher AES-CBC 256 bit keys</td>
<td>256 bits</td>
<td>N/A</td>
</tr>
<tr>
<td>RSASHA1_1024</td>
<td>Asymmetric encryption using block cipher RSA 1024 bit keys</td>
<td>1024 bits</td>
<td>SHA1</td>
</tr>
<tr>
<td>RSASHA1_2048</td>
<td>Asymmetric encryption using block cipher RSA 2048 bit keys</td>
<td>2048 bits</td>
<td>SHA1</td>
</tr>
<tr>
<td>RSASHA256_2048</td>
<td>Asymmetric encryption using block cipher RSA 2048 bit keys</td>
<td>2048 bits</td>
<td>SHA256</td>
</tr>
<tr>
<td>RSASHA256_3072</td>
<td>Asymmetric encryption using block cipher RSA 3072 bit keys</td>
<td>3072 bits</td>
<td>SHA256</td>
</tr>
</tbody>
</table>

typedef struct {
    UINT8     KeyIdentifierSize;
    VOID      *KeyIdentifier;
    EFI_GUID  KeyFormat;
    VOID      *KeyValue;
    EFI_STATUS KeyStatus;
} EFI_KMS_KEY_DESCRIPTOR;

KeyIdentifierSize The size of the KeyIdentifier field in bytes. This field is limited to the range 0 to 255.

KeyIdentifier Pointer to an array of KeyIdentifierType elements.

KeyFormat An EFI_GUID which specifies the algorithm and key value size for this key.

KeyValue Pointer to a key value for a key specified by the KeyFormat field. A NULL value for this field indicates that no key is available.

KeyStatus Specifies the results of KMS operations performed with this descriptor. This field is used to indicate the status of individual operations when a KMS function is called with multiple EFI_KMS_KEY_DESCRIPTOR structures. KeyStatus codes returned for the individual key requests are:
## Status Codes Returned

<table>
<thead>
<tr>
<th>Code</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>EFI_SUCCESS</td>
<td>Successfully processed this key.</td>
</tr>
<tr>
<td>EFI_WARN_STALE_DATA</td>
<td>Successfully processed this key, however, the key’s parameters exceed internal policies/limits and should be replaced.</td>
</tr>
<tr>
<td>EFI_COMPROMISED_DATA</td>
<td>Successfully processed this key, but the key may have been compromised and must be replaced.</td>
</tr>
<tr>
<td>EFI_UNSUPPORTED</td>
<td>Key format is not supported by the service.</td>
</tr>
<tr>
<td>EFI_OUT_OF_RESOURCES</td>
<td>Could not allocate resources for the key processing.</td>
</tr>
<tr>
<td>EFI_TIMEOUT</td>
<td>Timed out waiting for device or key server.</td>
</tr>
<tr>
<td>EFI_DEVICE_ERROR</td>
<td>Device or key server error.</td>
</tr>
<tr>
<td>EFI_INVALID_PARAMETER</td>
<td>KeyFormat is invalid.</td>
</tr>
<tr>
<td>EFI_NOT_FOUND</td>
<td>The key does not exist on the KMS.</td>
</tr>
</tbody>
</table>

```c
#define EFI_KMS_ATTRIBUTE_TYPE_NONE      0x00
#define EFI_KMS_ATTRIBUTE_TYPE_INTEGER    0x01
#define EFI_KMS_ATTRIBUTE_TYPE_LONG_INTEGER  0x02
#define EFI_KMS_ATTRIBUTE_TYPE_BIG_INTEGER  0x03
#define EFI_KMS_ATTRIBUTE_TYPE_ENUMERATION  0x04
#define EFI_KMS_ATTRIBUTE_TYPE_BOOLEAN    0x05
#define EFI_KMS_ATTRIBUTE_TYPE_BYTE_STRING  0x06
#define EFI_KMS_ATTRIBUTE_TYPE_TEXT_STRING  0x07
#define EFI_KMS_ATTRIBUTE_TYPE_DATE_TIME   0x08
#define EFI_KMS_ATTRIBUTE_TYPE_INTERVAL    0x09
#define EFI_KMS_ATTRIBUTE_TYPE_STRUCTURE   0x0A
#define EFI_KMS_ATTRIBUTE_TYPE_DYNAMIC    0x0B
```

typedef struct {
    UINT32             FieldCount;
    EFI_KMS_DYNAMIC_FIELD Field[1];
} EFI_KMS_DYNAMIC_ATTRIBUTE;

**FieldCount**

The number of members in the `EFI_KMS_DYNAMIC_ATTRIBUTE` structure.

**Field**

An array of `EFI_KMS_DYNAMIC_FIELD` structures.
typedef struct {
    UINT16  Tag;
    UINT16  Type;
    UINT32  Length;
    UINT8   KeyAttributeData[1];
} EFI_KMS_DYNAMIC_FIELD;

Tag           Part of a tag-type-length triplet that identifies the
              KeyAttributeData formatting. The definition of the value is
              outside the scope of this standard and may be defined by the
              KMS.

Type          Part of a tag-type-length triplet that identifies the
              KeyAttributeData formatting. The definition of the value is
              outside the scope of this standard and may be defined by the
              KMS.

Length        Length in bytes of the KeyAttributeData.

KeyAttributeData An array of bytes to hold the attribute data associated with
                  the KeyAttributeIdentifier.

typedef struct {
    UINT8   KeyAttributeIdentifierType;
    UINT8   KeyAttributeIdentifierCount;
    VOID    *KeyAttributeIdentifier;
    UINT16  KeyAttributeInstance;
    UINT16  KeyAttributeType;
    UINT16  KeyAttributeValueSize;
    VOID    *KeyAttributeValue;
    EFI_STATUS  KeyAttributeStatus;
} EFI_KMS_KEY_ATTRIBUTE;

KeyAttributeIdentifierType The data type used for the KeyAttributeIdentifier field.
                             Values for this field are defined by the EFI_KMS_DATA_TYPE
                             constants, except that EFI_KMS_DATA_TYPE_BINARY is not
                             valid for this field.

KeyAttributeIdentifierCount The length of the KeyAttributeIdentifier field in units
                               defined by KeyAttributeIdentifierType field. This field is
                               limited to the range 0 to 255.

KeyAttributeIdentifier Pointer to an array of KeyAttributeIdentifierType
                           elements. For string types, there must not be a null-
                           termination element at the end of the array.
KeyAttributeInstance
The instance number of this attribute. If there is only one instance, the value is set to one. If this value is set to 0xFFFF (all binary 1’s) then this field should be ignored if an output or treated as a wild card matching any value if it is an input. If the attribute is stored with this field, it will match any attribute request regardless of the setting of the field in the request. If set to 0xFFFF in the request, it will match any attribute with the same KeyAttributeIdentifier.

KeyAttributeType
The data type of the KeyAttributeValue (e.g. struct, bool, etc.). See the list of KeyAttributeType definitions.

KeyAttributeValueSize
The size in bytes of the KeyAttribute field. A value of zero for this field indicates that no key attribute value is available.

KeyAttributeValue
Pointer to a key attribute value for the attribute specified by the KeyAttributeIdentifier field. If the KeyAttributeValueSize field is zero, then this field must be NULL.

KeyAttributeStatus
Specifies the results of KMS operations performed with this attribute. This field is used to indicate the status of individual operations when a KMS function is called with multiple EFI_KMS_KEY_ATTRIBUTE structures. KeyAttributeStatus codes returned for the individual key attribute requests are:

<table>
<thead>
<tr>
<th>Status Code</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>EFI_SUCCESS</td>
<td>Successfully processed this request.</td>
</tr>
<tr>
<td>EFI_WARN_STALE_DATA</td>
<td>Successfully processed this request, however, the key’s parameters exceed internal policies/limits and should be replaced.</td>
</tr>
<tr>
<td>EFI_COMPROMISED_DATA</td>
<td>Successfully processed this request, but the key may have been compromised and must be replaced.</td>
</tr>
<tr>
<td>EFI_UNSUPPORTED</td>
<td>Key attribute format is not supported by the service.</td>
</tr>
<tr>
<td>EFI_OUT_OF_RESOURCES</td>
<td>Could not allocate resources for the request processing.</td>
</tr>
<tr>
<td>EFI_TIMEOUT</td>
<td>Timed out waiting for device or key server.</td>
</tr>
<tr>
<td>EFI_DEVICE_ERROR</td>
<td>Device or key server error.</td>
</tr>
<tr>
<td>EFI_INVALID_PARAMETER</td>
<td>A field in the EFI_KMS_KEY_ATTRIBUTE structure is invalid.</td>
</tr>
<tr>
<td>EFI_NOT_FOUND</td>
<td>The key attribute does not exist on the KMS.</td>
</tr>
</tbody>
</table>

Description
The EFI_KMS_SERVICE_PROTOCOL defines a UEFI protocol that can be used by UEFI drivers and applications to access cryptographic keys associated with their operation that are stored and possibly managed by a remote key management service (KMS). For example, a storage device driver may require a set of one or more keys to enable access to regions on the storage devices that it manages.

The protocol can be used to request the generation of new keys from the KMS, to register locally generated keys with the KMS, to retrieve existing keys from the KMS, and to delete obsolete keys from
the KMS. It also allows the device driver to manage attributes associated with individual keys on the KMS, and to retrieve keys based on those attributes.

A platform implementing this protocol may use internal or external key servers to provide the functionality required by this protocol. For external servers, the protocol implementation is expected to supply and maintain the connection parameters required to connect and authenticate to the remote server. The connection may be made during the initial installation of the protocol, or it may be delayed until the first `GetServiceStatus()` request is received.

Each client using the KMS protocol may identify itself to the protocol implementation using a `EFI_KMS_CLIENT_INFO` structure. If the KMS supported by this protocol requires the client to provide a client identifier, then this structure must be provided on all function calls.

While this protocol is intended to abstract the functions associated with storing and managing keys so that the protocol user does not have to be aware of the specific KMS providing the service, it can also be used by callers which must interact directly with a specific KMS. For these users, the protocol manages the connection to the KMS while the user controls the operational interface via a client data pass thru function.

The `EFI_KMS_SERVICE_PROTOCOL` provides the capability for the caller to pass arbitrary data to the KMS or to receive such data back from the KMS via parameters on most functions. The use of such data is at the discretion of the caller, but it should only be used sparingly as it reduces the interoperability of the caller’s software.

**EFI_KMS_PROTOCOL.GetServiceStatus()**

**Summary**

Get the current status of the key management service.

**Prototype**

```c
typedef EFI_STATUS
    (EFIAPI *EFI_KMS_GET_SERVICE_STATUS) (  
      IN EFI_KMS_PROTOCOL  *This
    );
```

**Parameters**

- `This` Pointer to the `EFI_KEY_MANAGEMENT_SERVICE_PROTOCOL` instance.

**Description**

The `GetServiceStatus()` function allows the user to query the current status of the KMS and should be called before attempting any operations to the KMS. If the protocol has not been marked as available, then the user must call this function to attempt to initiate the connection to the KMS as it may have been deferred to the first user by the system firmware.

If the connection to the KMS has not yet been established by the system firmware, then this function will attempt to establish the connection, update the protocol structure content as appropriate, and mark the service as available.
Status Codes Returned

<table>
<thead>
<tr>
<th>Status Code</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>EFI_SUCCESS</td>
<td>The KMS is ready for use.</td>
</tr>
<tr>
<td>EFI_NOT_READY</td>
<td>No connection to the KMS is available.</td>
</tr>
<tr>
<td>EFI_NO_MAPPING</td>
<td>No valid connection configuration exists for the KMS.</td>
</tr>
<tr>
<td>EFI_NO_RESPONSE</td>
<td>No response was received from the KMS.</td>
</tr>
<tr>
<td>EFI_DEVICE_ERROR</td>
<td>An error occurred when attempting to access the KMS.</td>
</tr>
<tr>
<td>EFI_INVALID_PARAMETER</td>
<td>This is NULL.</td>
</tr>
</tbody>
</table>

**EFI_KMS_PROTOCOL:RegisterClient()**

**Summary**

Register client information with the supported KMS.

**Prototype**

```c
typedef EFI_STATUS (EFIAPIC *EFI_KMS_PROTOCOL_API) (
  IN EFI_KMS_PROTOCOL *This,
  IN EFI_KMS_CLIENT_INFO *Client,
  IN OUT UINTN *ClientDataSize OPTIONAL,
  IN OUT VOID **ClientData OPTIONAL
);
```

**Parameters**

- **This**
  Pointer to the **EFI_KEY_MANAGEMENT_SERVICE_PROTOCOL** instance.

- **Client**
  Pointer to a valid **EFI_KMS_CLIENT_INFO** structure.

- **ClientDataSize**
  Pointer to the size, in bytes, of an arbitrary block of data specified by the **ClientData** parameter. This parameter may be **NULL**, in which case the **ClientData** parameter will be ignored and no data will be transferred to or from the KMS. If the parameter is not **NULL**, then **ClientData** must be a valid pointer. If the value pointed to is 0, no data will be transferred to the KMS, but data may be returned by the KMS. For all non-zero values **ClientData** will be transferred to the KMS, which may also return data to the caller. In all cases, the value upon return to the caller will be the size of the data block returned to the caller, which will be zero if no data is returned from the KMS.

- **ClientData**
  Pointer to a pointer to an arbitrary block of data of **ClientDataSize** that is to be passed directly to the KMS if it supports the use of client data. This parameter may be **NULL** if and only if the **ClientDataSize** parameter is also **NULL**. Upon return to the caller, **ClientData** points to a block of data of **ClientDataSize** that was returned from the KMS. If the
returned value for *ClientDataSize is zero, then the returned value for *ClientData must be NULL and should be ignored by the caller. The KMS protocol consumer is responsible for freeing all valid buffers used for client data regardless of whether they are allocated by the caller for input to the function or by the implementation for output back to the caller.

Description

The RegisterClient() function registers client information with the KMS using a EFI_KMS_CLIENT_INFO structure.

There are two methods of handling client information. The caller may supply a client identifier in the EFI_KMS_CLIENT_INFO structure prior to making the call along with an optional name string. The client identifier will be passed on to the KMS if it supports client identifiers. If the KMS accepts the client id, then the EFI_KMS_CLIENT_INFO structure will be returned to the caller unchanged. If the KMS does not accept the client id, it may simply reject the request, or it may supply an alternate identifier of its own,

The caller may also request a client identifier from the KMS by passing NULL values in the EFI_KMS_CLIENT_INFO structure. If the KMS supports this action, it will generate the identifier and return it in the structure. Otherwise, the implementation may generate a unique identifier, returning it in the structure, or it may indicate that the function is unsupported.

The ClientDataSize and ClientData parameters allow the caller to pass an arbitrary block of data to/from the KMS for uses such as auditing or access control. The KMS protocol implementation does not alter this data block other than to package it for transmission to the KMS. The use of these parameters is optional.
Status Codes Returned

<table>
<thead>
<tr>
<th>Status Code</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>EFI_SUCCESS</td>
<td>The client information has been accepted by the KMS.</td>
</tr>
<tr>
<td>EFI_NOT_READY</td>
<td>No connection to the KMS is available.</td>
</tr>
<tr>
<td>EFI_NO_RESPONSE</td>
<td>There was no response from the device or the key server.</td>
</tr>
<tr>
<td>EFI_ACCESS_DENIED</td>
<td>Access was denied by the device or the key server.</td>
</tr>
<tr>
<td>EFI_DEVICE_ERROR</td>
<td>An error occurred when attempting to access the KMS.</td>
</tr>
<tr>
<td>EFI_OUT_OF_RESOURCES</td>
<td>Required resources were not available to perform the function.</td>
</tr>
<tr>
<td>EFI_INVALID_PARAMETER</td>
<td>This is NULL.</td>
</tr>
<tr>
<td>EFI_UNSUPPORTED</td>
<td>The KMS does not support the use of client identifiers.</td>
</tr>
</tbody>
</table>

EFI_KMS_PROTOCOL.CreateKey()

Summary
Request that the KMS generate one or more new keys and associate them with key identifiers. The key value(s) is returned to the caller.

Prototype
```c
typedef
  EFI_STATUS
  (EFIAPI *EFI_KMS_CREATE_KEY) (    
    IN   EFI_KMS_PROTOCOL    *This, 
    IN   EFI_KMS_CLIENT_INFO *Client, 
    IN OUT UINT16            *KeyDescriptorCount, 
    IN OUT EFI_KMS_KEY_DESCRIPTOR *KeyDescriptors, 
    IN OUT UINTN              *ClientDataSize OPTIONAL, 
    IN OUT VOID               **ClientData OPTIONAL  
  );
```

Parameters
- **This**: Pointer to this EFI_KMS_PROTOCOL instance.
- **Client**: Pointer to a valid EFI_KMS_CLIENT_INFO structure.
- **KeyDescriptorCount**: Pointer to a count of the number of key descriptors to be processed by this operation. On return, this number will be updated with the number of key descriptors successfully processed.
- **KeyDescriptors**: Pointer to an array of EFI_KMS_KEY_DESCRIPTOR structures which describe the keys to be generated.

On input, the KeyIdentifierSize and the KeyIdentifier may specify an identifier to be used for the key, but this is not required. The KeyFormat field must specify a key format GUID reported as supported by the KeyFormats field of the EFI_KMS_PROTOCOL. The value for this field in the first key descriptor will be considered the default value for subsequent
key descriptors requested in this operation if those key
descriptors have a NULL GUID in the key format field.

On output, the KeyIdentifierSize and KeyIdentifier
fields will specify an identifier for the key which will be either
the original identifier if one was provided, or an identifier
generated either by the KMS or the KMS protocol
implementation. The KeyFormat field will be updated with the
GUID used to generate the key if it was a NULL GUID, and the
KeyValue field will contain a pointer to memory containing
the key value for the generated key. Memory for both the
KeyIdentifier and the KeyValue fields will be allocated with
the BOOT_SERVICES_DATA type and must be freed by the
caller when it is no longer needed. Also, the KeyStatus field
must reflect the result of the request relative to that key.

ClientDataSize

Pointer to the size, in bytes, of an arbitrary block of data
specified by the ClientData parameter. This parameter may be
NULL, in which case the ClientData parameter will be
ignored and no data will be transferred to or from the KMS. If
the parameter is not NULL, then ClientData must be a valid
pointer. If the value pointed to is 0, no data will be transferred
to the KMS, but data may be returned by the KMS. For all non-
zero values *ClientData will be transferred to the KMS,
which may also return data to the caller. In all cases, the value
upon return to the caller will be the size of the data block
returned to the caller, which will be zero if no data is returned
from the KMS.

ClientData

Pointer to a pointer to an arbitrary block of data of
*ClientDataSize that is to be passed directly to the KMS if it
supports the use of client data. This parameter may be NULL if
and only if the ClientDataSize parameter is also NULL. Upon
return to the caller, *ClientData points to a block of data of
*ClientDataSize that was returned from the KMS. If the
returned value for *ClientDataSize is zero, then the returned
value for *ClientData must be NULL and should be ignored
by the caller. The KMS protocol consumer is responsible for
freeing all valid buffers used for client data regardless of
whether they are allocated by the caller for input to the
function or by the implementation for output back to the
caller.

Description

The CreateKey() method requests the generation of one or more new keys, and key
identifier and key values are returned to the caller. The support of this function is optional
as some key servers do not provide a key generation capability.

The Client parameter identifies the caller to the key management service. This identifier
may be used for auditing or access control. This parameter is optional unless the KMS
requires a client identifier in order to perform the requested action.
The `KeyDescriptorCount` and `KeyDescriptors` parameters are used to specify the key algorithm, size, and attributes for the requested keys. Any number of keys may be requested in a single operation, regardless of whether the KMS supports multiple key definitions in a single request or not. The KMS protocol implementation is responsible for generating the appropriate requests (single/multiple) to the KMS.

The `ClientDataSize` and `ClientData` parameters allow the caller to pass an arbitrary block of data to/from the KMS for uses such as auditing or access control. The KMS protocol implementation does not alter this data block other than to package it for transmission to the KMS. The use of these parameters is optional.

### Status Codes Returned

The `CreateKey()` function will return a status which indicates the overall status of the request. Note that this may be different from the status reported for individual key requests.

<table>
<thead>
<tr>
<th>Status Code</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>EFI_SUCCESS</strong></td>
<td>Successfully generated and retrieved all requested keys.</td>
</tr>
<tr>
<td><strong>EFI_UNSUPPORTED</strong></td>
<td>This function is not supported by the KMS.</td>
</tr>
<tr>
<td></td>
<td>--OR-- One (or more) of the key requests submitted is not supported by the</td>
</tr>
<tr>
<td></td>
<td>KMS. Check individual key request(s) to see which ones may have been</td>
</tr>
<tr>
<td></td>
<td>processed.</td>
</tr>
<tr>
<td><strong>EFI_OUT_OF_RESOURCES</strong></td>
<td>Required resources were not available for the operation.</td>
</tr>
<tr>
<td><strong>EFI_TIMEOUT</strong></td>
<td>Timed out waiting for device or key server. Check individual key request(s)</td>
</tr>
<tr>
<td></td>
<td>to see which ones may have been processed.</td>
</tr>
<tr>
<td><strong>EFI_ACCESS_DENIED</strong></td>
<td>Access was denied by the device or the key server; OR a <code>ClientId</code> is required</td>
</tr>
<tr>
<td></td>
<td>by the server and either no id was provided or an invalid id was provided</td>
</tr>
<tr>
<td><strong>EFI_DEVICE_ERROR</strong></td>
<td>An error occurred when attempting to access the KMS. Check individual key</td>
</tr>
<tr>
<td></td>
<td>request(s) to see which ones may have been processed.</td>
</tr>
<tr>
<td><strong>EFI_INVALID_PARAMETER</strong></td>
<td>This is <code>NULL</code>, <code>ClientId</code> is required but it is <code>NULL</code>, <code>KeyDescriptorCount</code> is <code>NULL</code>, or <code>Keys</code> is <code>NULL</code></td>
</tr>
<tr>
<td><strong>EFI_NOT_FOUND</strong></td>
<td>One or more <code>EFI_KMS_KEY_DESCRIPTOR</code> structures could not be processed</td>
</tr>
<tr>
<td></td>
<td>properly. <code>KeyDescriptorCount</code> contains the number of structures which were</td>
</tr>
<tr>
<td></td>
<td>successfully processed. Individual structures will reflect the status of the</td>
</tr>
<tr>
<td></td>
<td>processing for that structure.</td>
</tr>
</tbody>
</table>

### EFI_KMS_PROTOCOL.GetKey()

**Summary**

Retrieve an existing key.
Prototype

typedef
    EFI_STATUS
    (EFIAPIC *EFI_KMS_GET_KEY) (  
        IN   EFI_KMS_PROTOCOL     *This,
        IN   EFI_KMS_CLIENT_INFO  *Client,
        IN OUT UINT16             *KeyDescriptorCount,
        IN OUT EFI_KMS_KEY_DESCRIPTOR *KeyDescriptors,
        IN OUT UINTN              *ClientDataSize OPTIONAL,
        IN OUT VOID               **ClientData OPTIONAL);

Parameters

This          Pointer to this EFI_KMS_PROTOCOL instance.
Client        Pointer to a valid EFI_KMS_CLIENT_INFO structure.
KeyDescriptorCount Pointer to a count of the number of keys to be processed by this operation. On return, this number will be updated with number of keys successfully processed.

KeyDescriptors Pointer to an array of EFI_KMS_KEY_DESCRIPTOR structures which describe the keys to be retrieved from the KMS. On input, the KeyIdentifierSize and the KeyIdentifier must specify an identifier to be used to retrieve a specific key. All other fields in the descriptor should be NULL. On output, the KeyIdentifierSize and KeyIdentifier fields will be unchanged, while the KeyFormat and KeyValue fields will be updated values associated with this key identifier. Memory for the KeyValue field will be allocated with the BOOT_SERVICES_DATA type and must be freed by the caller when it is no longer needed. Also, the KeyStatus field will reflect the result of the request relative to the individual key descriptor.

ClientDataSize Pointer to the size, in bytes, of an arbitrary block of data specified by the ClientData parameter. This parameter may be NULL, in which case the ClientData parameter will be ignored and no data will be transferred to or from the KMS. If the parameter is not NULL, then ClientData must be a valid pointer. If the value pointed to is 0, no data will be transferred to the KMS, but data may be returned by the KMS. For all non-zero values *ClientData will be transferred to the KMS, which may also return data to the caller. In all cases, the value upon return to the caller will be the size of the data block returned to the caller, which will be zero if no data is returned from the KMS.

ClientData Pointer to a pointer to an arbitrary block of data of *ClientDataSize that is to be passed directly to the KMS if it supports the use of client data. This parameter may be NULL if and only if the ClientDataSize parameter is also NULL. Upon return to the caller, *ClientData points to a block of data of
*ClientDataSize* that was returned from the KMS. If the returned value for *ClientDataSize* is zero, then the returned value for *ClientData* must be **NULL** and should be ignored by the caller. The KMS protocol consumer is responsible for freeing all valid buffers used for client data regardless of whether they are allocated by the caller for input to the function or by the implementation for output back to the caller.

**Description**

The **GetKey()** function retrieves one or more existing keys from the KMS and returns the key values to the caller. This function must be supported by every KMS protocol instance.

The **Client** parameter identifies the caller to the key management service. It may be used for auditing or access control. The use of this parameter is optional unless the KMS requires it in order to perform the requested action.

The **KeyDescriptorCount** and **KeyDescriptors** parameters are used to specify the identifier(s) to be used to retrieve the key values, which will be returned in the **KeyFormat** and **KeyValue** fields of each **EFI_KMS_KEY_DESCRIPTOR** structure. Any number of keys may be requested in a single operation, regardless of whether the KMS supports multiple key definitions in a single request or not. The KMS protocol implementation is responsible for generating the appropriate requests (single/multiple) to the KMS.

The **ClientDataSize** and **ClientData** parameters allow the caller to pass an arbitrary block of data to/from the KMS for uses such as auditing or access control. The KMS protocol implementation does not alter this data block other than to package it for transmission to the KMS. The use of these parameters is optional.

**Status Codes Returned**

The **GetKey()** function will return a status which indicates the overall status of the request. Note that this may be different from the status reported for individual key requests.

<table>
<thead>
<tr>
<th>Status Code</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>EFI_SUCCESS</strong></td>
<td>Successfully retrieved all requested keys.</td>
</tr>
<tr>
<td><strong>EFI_OUT_OF_RESOURCES</strong></td>
<td>Could not allocate resources for the method processing.</td>
</tr>
<tr>
<td><strong>EFI_TIMEOUT</strong></td>
<td>Timed out waiting for device or key server. Check individual key request(s) to see which ones may have been processed.</td>
</tr>
<tr>
<td><strong>EFI_BUFFER_TOO_SMALL</strong></td>
<td>If multiple keys are associated with a single identifier, and the <strong>KeyValue</strong> buffer does not contain enough structures (<strong>KeyDescriptorCount</strong>) to contain all the key data, then the available structures will be filled and <strong>KeyDescriptorCount</strong> will be updated to indicate the number of keys which could not be processed.</td>
</tr>
<tr>
<td><strong>EFI_ACCESS_DENIED</strong></td>
<td>Access was denied by the device or the key server; OR a <strong>ClientId</strong> is required by the server and either none or an invalid id was provided</td>
</tr>
<tr>
<td><strong>EFI_DEVICE_ERROR</strong></td>
<td>Device or key server error. Check individual key request(s) to see which ones may have been processed.</td>
</tr>
<tr>
<td><strong>EFI_INVALID_PARAMETER</strong></td>
<td>This is <strong>NULL</strong>, <strong>ClientId</strong> is required but it is <strong>NULL</strong>, <strong>KeyDescriptorCount</strong> is <strong>NULL</strong>, or <strong>Keys</strong> is <strong>NULL</strong></td>
</tr>
</tbody>
</table>
**EFI_KMS_PROTOCOL.AddKey()**

**Summary**

Add a new key.

**Prototype**

```c
typedef EFI_STATUS (EFIAPI *EFI_KMS_PROTOCOL) (EFI_KMS_PROTOCOL *This,
IN EFI_KMS_PROTOCOL *This,
IN EFI_KMS_CLIENT_INFO *Client,
IN OUT UINT16 *KeyDescriptorCount,
IN OUT EFI_KMS_KEY_DESCRIPTOR *KeyDescriptors,
IN OUT UINTN *ClientDataSize OPTIONAL,
IN OUT VOID **ClientData OPTIONAL
);
```

**Parameters**

- **This**: Pointer to this `EFI_KMS_PROTOCOL` instance.
- **Client**: Pointer to a valid `EFI_KMS_CLIENT_INFO` structure.
- **KeyDescriptorCount**: Pointer to a count of the number of keys to be processed by this operation. On normal returns, this number will be updated with number of keys successfully processed.
- **KeyDescriptors**: Pointer to an array of `EFI_KMS_KEY_DESCRIPTOR` structures which describe the keys to be added. On input, the `KeyId` field for first key must contain valid identifier data to be used for adding a key to the KMS. The values for these fields in this key definition will be considered default values for subsequent keys requested in this operation. A value of 0 in any subsequent `KeyId` field will be replaced with the current default value. The `KeyFormat` and `KeyValue` fields for each key to be added must contain consistent values to be associated with the given `KeyId`. On return, the `KeyStatus` field will reflect the result of the operation for each key request.
- **ClientDataSize**: Pointer to the size, in bytes, of an arbitrary block of data specified by the `ClientData` parameter. This parameter may be `NULL`, in which case the `ClientData` parameter will be ignored and no data will be transferred to or from the KMS. If the parameter is not `NULL`, then `ClientData` must be a valid
pointer. If the value pointed to is 0, no data will be transferred to the KMS, but data may be returned by the KMS. For all non-zero values *ClientData will be transferred to the KMS, which may also return data to the caller. In all cases, the value upon return to the caller will be the size of the data block returned to the caller, which will be zero if no data is returned from the KMS.

**ClientData**

Pointer to a pointer to an arbitrary block of data of *ClientDataSize that is to be passed directly to the KMS if it supports the use of client data. This parameter may be **NULL** if and only if the *ClientDataSize parameter is also **NULL**. Upon return to the caller, *ClientData points to a block of data of *ClientDataSize that was returned from the KMS. If the returned value for *ClientDataSize is zero, then the returned value for *ClientData must be **NULL** and should be ignored by the caller. The KMS protocol consumer is responsible for freeing all valid buffers used for client data regardless of whether they are allocated by the caller for input to the function or by the implementation for output back to the caller.

**Description**

The **AddKey()** function registers a new key with the key management service. The support for this method is optional, as not all key servers support importing keys from clients.

The **Client** parameter identifies the caller to the key management service. It may be used for auditing or access control. The use of this parameter is optional unless the KMS requires it in order to perform the requested action.

The **KeyDescriptorCount** and **KeyDescriptors** parameters are used to specify the key identifier, key format and key data to be registered on the. Any number of keys may be registered in a single operation, regardless of whether the KMS supports multiple key definitions in a single request or not. The KMS protocol implementation is responsible for generating the appropriate requests (single/multiple) to the KMS.

The **ClientDataSize** and **ClientData** parameters allow the caller to pass an arbitrary block of data to/from the KMS for uses such as auditing or access control. The KMS protocol implementation does not alter this data block other than to package it for transmission to the KMS. The use of these parameters is optional.

**Status Codes Returned**

The **AddKey()** function will return a status which indicates the overall status of the request. Note that this may be different from the status reported for individual key requests.

<table>
<thead>
<tr>
<th>Status Code</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>EFI_SUCCESS</td>
<td>Successfully added all requested keys.</td>
</tr>
<tr>
<td>EFI_OUT_OF_RESOURCES</td>
<td>Could not allocate required resources.</td>
</tr>
<tr>
<td>EFI_TIMEOUT</td>
<td>Timed out waiting for device or key server. Check individual key request(s) to see which ones may have been processed.</td>
</tr>
</tbody>
</table>
## EFI_KMS_PROTOCOL.DeleteKey()

### Summary
Delete an existing key from the KMS database.

### Prototype
```c
typedef
EFI_STATUS
(EIFIAPI *EFI_KMS_DELETE_KEY) (
    IN EFI_KMS_PROTOCOL   *This,
    IN   EFI_KMS_CLIENT_INFO  *Client,
    IN OUT UINT16        *KeyDescriptorCount,
    IN OUT EFI_KMS_KEY_DESCRIPTOR *KeyDescriptors,
    IN OUT UINTN         *ClientDataSize OPTIONAL,
    IN OUT VOID          **ClientData OPTIONAL
);
```

### Parameters
- **This**: Pointer to this EFI_KMS_PROTOCOL instance.
- **Client**: Pointer to a valid EFI_KMS_CLIENT_INFO structure.
- **KeyDescriptorCount**: Pointer to a count of the number of keys to be processed by this operation. On normal returns, this number will be updated with number of keys successfully processed.
- **KeyDescriptors**: Pointer to an array of EFI_KMS_KEY_DESCRIPTOR structures which describe the keys to be deleted. On input, the KeyId field for first key must contain valid identifier data to be used for adding a key to the KMS. The values for these fields in this table:

<table>
<thead>
<tr>
<th>Error Code</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>EFI_BUFFER_TOO_SMALL</td>
<td>If multiple keys are associated with a single identifier, and the KeyValue buffer does not contain enough structures (KeyDescriptorCount) to contain all the key data, then the available structures will be filled and KeyDescriptorCount will be updated to indicate the number of keys which could not be processed.</td>
</tr>
<tr>
<td>EFI_ACCESS_DENIED</td>
<td>Access was denied by the device or the key server; OR a ClientId is required by the server and either none or an invalid id was provided.</td>
</tr>
<tr>
<td>EFI_DEVICE_ERROR</td>
<td>Device or key server error. Check individual key request(s) to see which ones may have been processed.</td>
</tr>
<tr>
<td>EFI_INVALID_PARAMETER</td>
<td>This is NULL, ClientId is required but it is NULL, KeyDescriptorCount is NULL, or Keys is NULL</td>
</tr>
<tr>
<td>EFI_NOT_FOUND</td>
<td>One or more EFI_KMS_KEY_DESCRIPTOR structures could not be processed properly. KeyDescriptorCount contains the number of structures which were successfully processed. Individual structures will reflect the status of the processing for that structure.</td>
</tr>
<tr>
<td>EFI_UNSUPPORTED</td>
<td>The implementation/KMS does not support this function.</td>
</tr>
</tbody>
</table>
key definition will be considered default values for subsequent keys requested in this operation. A value of 0 in any subsequent KeyId field will be replaced with the current default value. The KeyFormat and KeyValue fields are ignored, but should be 0. On return, the KeyStatus field will reflect the result of the operation for each key request.

**ClientDataSize**  
Pointer to the size, in bytes, of an arbitrary block of data specified by the ClientData parameter. This parameter may be NULL, in which case the ClientData parameter will be ignored and no data will be transferred to or from the KMS. If the parameter is not NULL, then ClientData must be a valid pointer. If the value pointed to is 0, no data will be transferred to the KMS, but data may be returned by the KMS. For all non-zero values *ClientData will be transferred to the KMS, which may also return data to the caller. In all cases, the value upon return to the caller will be the size of the data block returned to the caller, which will be zero if no data is returned from the KMS.

**ClientData**  
Pointer to a pointer to an arbitrary block of data of *ClientDataSize that is to be passed directly to the KMS if it supports the use of client data. This parameter may be NULL if and only if the ClientDataSize parameter is also NULL. Upon return to the caller, *ClientData points to a block of data of *ClientDataSize that was returned from the KMS. If the returned value for *ClientDataSize is zero, then the returned value for *ClientData must be NULL and should be ignored by the caller. The KMS protocol consumer is responsible for freeing all valid buffers used for client data regardless of whether they are allocated by the caller for input to the function or by the implementation for output back to the caller.

**Description**

The **DeleteKey()** function deregisters an existing key from the device or KMS. The support for this method is optional, as not all key servers support deleting keys from clients.

The **Client** parameter identifies the caller to the key management service. It may be used for auditing or access control. The use of this parameter is optional unless the KMS requires it in order to perform the requested action.

The **KeyDescriptorCount** and **KeyDescriptors** parameters are used to specify the key identifier(s) for the keys to be deleted. Any number of keys may be deleted in a single operation, regardless of whether the KMS supports multiple key definitions in a single request or not. The KMS protocol implementation is responsible for generating the appropriate requests (single/multiple) to the KMS.

The **ClientDataSize** and **ClientData** parameters allow the caller to pass an arbitrary block of data to/from the KMS for uses such as auditing or access control. The KMS protocol implementation does not alter this data block other than to package it for transmission to the KMS. The use of these parameters is optional.
Status Codes Returned

The `DeleteKey()` function will return a status which indicates the overall status of the request. Note that this may be different from the status reported for individual key requests.

<table>
<thead>
<tr>
<th>Code</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>EFI_SUCCESS</td>
<td>Successfully deleted all requested keys.</td>
</tr>
<tr>
<td>EFI_OUT_OF_RESOURCES</td>
<td>Could not allocate required resources.</td>
</tr>
<tr>
<td>EFI_TIMEOUT</td>
<td>Timed out waiting for device or key server. Check individual key request(s) to see which ones may have been processed.</td>
</tr>
<tr>
<td>EFI_ACCESS_DENIED</td>
<td>Access was denied by the device or the key server; OR a <code>ClientId</code> is required by the server and either none or an invalid id was provided</td>
</tr>
<tr>
<td>EFI_DEVICE_ERROR</td>
<td>Device or key server error. Check individual key request(s) to see which ones may have been processed.</td>
</tr>
<tr>
<td>EFI_INVALID_PARAMETER</td>
<td>This is <code>NULL</code>, <code>ClientId</code> is required but it is <code>NULL</code>, <code>KeyDescriptorCount</code> is <code>NULL</code>, or <code>Keys</code> is <code>NULL</code>.</td>
</tr>
<tr>
<td>EFI_NOT_FOUND</td>
<td>One or more <code>EFI_KMS_KEY_DESCRIPTOR</code> structures could not be processed properly. <code>KeyDescriptorCount</code> contains the number of structures which were successfully processed. Individual structures will reflect the status of the processing for that structure.</td>
</tr>
<tr>
<td>EFI_UNSUPPORTED</td>
<td>The implementation/KMS does not support this function.</td>
</tr>
</tbody>
</table>

`EFI_KMS_PROTOCOL.GetKeyAttributes()`

Summary

Get one or more attributes associated with a specified key identifier. If none are found, the returned attributes count contains a value of zero.

Prototype

```c
typedef EFI_STATUS
    (EFIAPI *EFI_KMS_GET_KEY_ATTRIBUTES) (    
    IN   EFI_KMS_PROTOCOL       *This,  
    IN   EFI_KMS_CLIENT_INFO   *Client,  
    IN   UINT8                 *KeyIdentifierSize,  
    IN   CONST VOID            *KeyIdentifier,  
    IN OUT UINT16              *KeyAttributesCount,  
    IN OUT EFI_KMS_KEY_ATTRIBUTE *KeyAttributes,  
    IN OUT UINTN               *ClientDataSize OPTIONAL,  
    IN OUT VOID                **ClientData OPTIONAL);
```

Parameters

- `This` Pointer to this `EFI_KMS_PROTOCOL` instance.
- `Client` Pointer to a valid `EFI_KMS_CLIENT_INFO` structure.
- `KeyIdentifierSize` Pointer to the size in bytes of the `KeyIdentifier` variable.
KeyIdentifier: Pointer to the key identifier associated with this key.

KeyAttributesCount: Pointer to the number of EFI_KMS_KEY_ATTRIBUTE structures associated with the Key identifier. If none are found, the count value is zero on return. On input this value reflects the number of KeyAttributes that may be returned. On output, the value reflects the number of completed KeyAttributes structures found.

KeyAttributes: Pointer to an array of EFI_KMS_KEYgetAttribute structures associated with the Key Identifier. On input, the fields in the structure should be NULL. On output, the attribute fields will have updated values for attributes associated with this key identifier.

ClientDataSize: Pointer to the size, in bytes, of an arbitrary block of data specified by the ClientData parameter. This parameter may be NULL, in which case the ClientData parameter will be ignored and no data will be transferred to or from the KMS. If the parameter is not NULL, then ClientData must be a valid pointer. If the value pointed to is 0, no data will be transferred to the KMS, but data may be returned by the KMS. For all non-zero values *ClientData will be transferred to the KMS, which may also return data to the caller. In all cases, the value upon return to the caller will be the size of the data block returned to the caller, which will be zero if no data is returned from the KMS.

ClientData: Pointer to a pointer to an arbitrary block of data of *ClientDataSize that is to be passed directly to the KMS if it supports the use of client data. This parameter may be NULL if and only if the ClientDataSize parameter is also NULL. Upon return to the caller, *ClientData points to a block of data of *ClientDataSize that was returned from the KMS. If the returned value for *ClientDataSize is zero, then the returned value for *ClientData must be NULL and should be ignored by the caller. The KMS protocol consumer is responsible for freeing all valid buffers used for client data regardless of whether they are allocated by the caller for input to the function or by the implementation for output back to the caller.

Description

The GetKeyAttributes() function returns one or more attributes for a key.

The ClientIdentifierSize and ClientIdentifier parameters identify the caller to the key management service. It may be used for auditing or access control. The use of this parameter is optional unless the KMS requires it in order to perform the requested action.

The KeyIdentifierSize and KeyIdentifier parameters identify the key whose attributes are to be returned by the key management service. They may be used to retrieve additional information about a key, whose format is defined by the KeyAttribute. Attributes returned may be of the same or different names.
The **ClientDataSize** and **ClientData** parameters allow the caller to pass an arbitrary block of data to/from the KMS for uses such as auditing or access control. The KMS protocol implementation does not alter this data block other than to package it for transmission to the KMS. The use of these parameters is optional unless the KMS requires it in order to perform the requested action.

### Status Codes Returned

The **GetKeyAttributes()** function will return a status which indicates the overall status of the request. Note that this may be different from the status reported for individual key attribute requests.

<table>
<thead>
<tr>
<th>Status Code</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>EFI_SUCCESS</td>
<td>Successfully retrieved all key attributes.</td>
</tr>
<tr>
<td>EFI_OUT_OF_RESOURCES</td>
<td>Could not allocate resources for the method processing.</td>
</tr>
<tr>
<td>EFI_TIMEOUT</td>
<td>Timed out waiting for device or key server. Check individual key attribute request(s) to see which ones may have been processed.</td>
</tr>
<tr>
<td>EFI_BUFFER_TOO_SMALL</td>
<td>If multiple key attributes are associated with a single identifier, and the <strong>KeyAttributes</strong> buffer does not contain enough structures (<strong>KeyAttributesCount</strong>) to contain all the key attributes data, then the available structures will be filled and <strong>KeyAttributesCount</strong> will be updated to indicate the number of key attributes which could not be processed.</td>
</tr>
<tr>
<td>EFI_ACCESS_DENIED</td>
<td>Access was denied by the device or the key server; OR a <strong>ClientId</strong> is required by the server and either none or an invalid id was provided</td>
</tr>
<tr>
<td>EFI_DEVICE_ERROR</td>
<td>Device or key server error. Check individual key attribute request(s) (i.e., key attribute status for each) to see which ones may have been processed.</td>
</tr>
<tr>
<td>EFI_INVALID_PARAMETER</td>
<td>This is <strong>NULL</strong>, <strong>ClientId</strong> is required but it is <strong>NULL</strong>, <strong>KeyIdentifierSize</strong> is <strong>NULL</strong>, or <strong>KeyIdentifier</strong> is <strong>NULL</strong>, or <strong>KeyAttributes</strong> is <strong>NULL</strong>, or <strong>KeyAttributesSize</strong> is <strong>NULL</strong></td>
</tr>
<tr>
<td>EFI_NOT_FOUND</td>
<td>The <strong>KeyIdentifier</strong> could not be found. <strong>KeyAttributesCount</strong> contains zero. Individual structures will reflect the status of the processing for that structure.</td>
</tr>
<tr>
<td>EFI_UNSUPPORTED</td>
<td>The implementation/KMS does not support this function</td>
</tr>
</tbody>
</table>

**EFI_KMS_PROTOCOL.AddKeyAttributes()**

**Summary**

Add one or more attributes to a key specified by a key identifier.
Prototype

```c
typedef
    EFI_STATUS
    (EFI_APPI *EFI_KMS_ADD_KEY_ATTRIBUTES) (
    IN   EFI_KMS_PROTOCOL   *This,
    IN   EFI_KMS_CLIENT_INFO *Client,
    IN   UINT               *KeyIdentifierSize,
    IN   CONST VOID         *KeyIdentifier,
    IN OUT UINT16           *KeyAttributesCount,
    IN OUT EFI_KMS_KEY_ATTRIBUTE *KeyAttributes,
    IN OUT UINTN            *ClientDataSize OPTIONAL,
    IN OUT VOID             **ClientData OPTIONAL);
```

Parameters

- **This**: Pointer to this `EFI_KMS_PROTOCOL` instance.
- **Client**: Pointer to a valid `EFI_KMS_CLIENT_INFO` structure.
- **KeyIdentifierSize**: Pointer to the size in bytes of the `KeyIdentifier` variable.
- **KeyIdentifier**: Pointer to the key identifier associated with this key.
- **KeyAttributesCount**: Pointer to the number of `EFI_KMS_KEY_ATTRIBUTE` structures to associate with the Key. On normal returns, this number will be updated with the number of key attributes successfully processed.
- **KeyAttributes**: Pointer to an array of `EFI_KMS_KEY_ATTRIBUTE` structures providing the attribute information to associate with the key. On input, the values for the fields in the structure are completely filled in. On return the `KeyAttributeStatus` field will reflect the result of the operation for each key attribute request.
- **ClientDataSize**: Pointer to the size, in bytes, of an arbitrary block of data specified by the `ClientData` parameter. This parameter may be `NULL`, in which case the `ClientData` parameter will be ignored and no data will be transferred to or from the KMS. If the parameter is not `NULL`, then `ClientData` must be a valid pointer. If the value pointed to is 0, no data will be transferred to the KMS, but data may be returned by the KMS. For all non-zero values `*ClientData` will be transferred to the KMS, which may also return data to the caller. In all cases, the value upon return to the caller will be the size of the data block returned to the caller, which will be zero if no data is returned from the KMS.
- **ClientData**: Pointer to a pointer to an arbitrary block of data of `*ClientDataSize` that is to be passed directly to the KMS if it supports the use of client data. This parameter may be `NULL` if and only if the `ClientDataSize` parameter is also `NULL`. Upon return to the caller, `*ClientData` points to a block of data of `*ClientDataSize` that was returned from the KMS. If the
returned value for *ClientDataSize is zero, then the returned value for *ClientData must be NULL and should be ignored by the caller. The KMS protocol consumer is responsible for freeing all valid buffers used for client data regardless of whether they are allocated by the caller for input to the function or by the implementation for output back to the caller.

Description

The AddKeyAttributes() function adds one or more key attributes. If this function is not supported by a KMS protocol instance then it is assumed that there is an alternative means available for attribute management in the KMS.

The Client parameters identify the caller to the key management service. It may be used for auditing or access control. The use of this parameter is optional unless the KMS requires it in order to perform the requested action.

The KeyIdentifierSize and KeyIdentifier parameters identify the key whose attributes are to be modified by the key management service.

The KeyAttributesCount and KeyAttributes parameters are used to specify the key attributes data to be registered on the KMS. Any number of attributes may be registered in a single operation, regardless of whether the KMS supports multiple key attribute definitions in a single request or not. The KMS protocol implementation is responsible for generating the appropriate requests (single/multiple) to the KMS. In certain error situations, the status of each attribute is updated indicating if that attribute was successfully registered or not.

The ClientDataSize and ClientData parameters allow the caller to pass an arbitrary block of data to/from the KMS for uses such as auditing or access control. The KMS protocol implementation does not alter this data block other than to package it for transmission to the KMS. The use of these parameters is optional unless the KMS requires it in order to perform the requested action.

Status Codes Returned

The AddKeyAttributes() function will return a status which indicates the overall status of the request. Note that this may be different from the status reported for individual key attribute requests. Status codes returned for AddKeyAttributes() are:

<table>
<thead>
<tr>
<th>Status Code</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>EFI_SUCCESS</td>
<td>Successfully added all requested key attributes.</td>
</tr>
<tr>
<td>EFI_OUT_OF_RESOURCES</td>
<td>Could not allocate required resources.</td>
</tr>
<tr>
<td>EFI_TIMEOUT</td>
<td>Timed out waiting for device or key server. Check individual key attribute request(s) to see which ones may have been processed.</td>
</tr>
<tr>
<td>EFI_BUFFER_TOO_SMALL</td>
<td>If multiple keys attributes are associated with a single key identifier, and the attributes buffer does not contain enough structures (KeyAttributesCount) to contain all the data, then the available structures will be filled and KeyAttributesCount will be updated to indicate the number of key attributes which could not be processed. The status of each key attribute is also updated indicating success or failure for that attribute in case there are other errors for those attributes that could be processed.</td>
</tr>
</tbody>
</table>
EFI_KMS_PROTOCOL.DeleteKeyAttributes()

Summary
Delete attributes to a key specified by a key identifier.

Prototype

define

typedef

(EFI_STATUS EFI_KMS_PROTOCOL *This,
 IN EFI_KMS_CLIENT_INFO *Client,
 IN UINT8 KeyIdentifierSize,
 IN CONST VOID *KeyIdentifier,
 IN OUT UINT16 KeyAttributesCount,
 IN OUT EFI_KMS_KEY_ATTRIBUTE *KeyAttributes,
 IN OUT UINTN ClientDataSize OPTIONAL,
 IN OUT VOID **ClientData OPTIONAL)

Parameters

This Pointer to this EFI_KMS_PROTOCOL instance.
Client Pointer to a valid EFI_KMS_CLIENT_INFO structure.
KeyIdentifierSize Pointer to the size in bytes of the KeyIdentifier variable.
KeyIdentifier Pointer to the key identifier associated with this key.
KeyAttributesCount Pointer to the number of EFI_KMS_KEY_ATTRIBUTE structures associated with the Key. On input, the count value is one or more. On normal returns, this number will be updated with the number of key attributes successfully processed.
KeyAttributes

Pointer to an array of `EFI_KMS_KEY_ATTRIBUTE` structures associated with the key. On input, the values for the fields in the structure are completely filled in. On return the `KeyAttributeStatus` field will reflect the result of the operation for each key attribute request.

ClientDataSize

Pointer to the size, in bytes, of an arbitrary block of data specified by the `ClientData` parameter. This parameter may be `NULL`, in which case the `ClientData` parameter will be ignored and no data will be transferred to or from the KMS. If the parameter is not `NULL`, then `ClientData` must be a valid pointer. If the value pointed to is 0, no data will be transferred to the KMS, but data may be returned by the KMS. For all non-zero values `*ClientData` will be transferred to the KMS, which may also return data to the caller. In all cases, the value upon return to the caller will be the size of the data block returned to the caller, which will be zero if no data is returned from the KMS.

ClientData

Pointer to a pointer to an arbitrary block of data of `*ClientDataSize` that is to be passed directly to the KMS if it supports the use of client data. This parameter may be `NULL` if and only if the `ClientDataSize` parameter is also `NULL`. Upon return to the caller, `*ClientData` points to a block of data of `*ClientDataSize` that was returned from the KMS. If the returned value for `*ClientDataSize` is zero, then the returned value for `*ClientData` must be `NULL` and should be ignored by the caller. The KMS protocol consumer is responsible for freeing all valid buffers used for client data regardless of whether they are allocated by the caller for input to the function or by the implementation for output back to the caller.

Description

The `DeleteKeyAttributes()` function removes key attributes for a key with the key management service.

The `Client` parameter identifies the caller to the key management service. It may be used for auditing or access control. The use of this parameter is optional unless the KMS requires it in order to perform the requested action.

The `KeyIdentifierSize` and `KeyIdentifier` parameters identify the key whose attributes are to be modified by the key management service.

The `KeyAttributesCount` and `KeyAttributes` parameters are used to specify the key attributes data to be deleted on the KMS. Any number of attributes may be deleted in a single operation, regardless of whether the KMS supports multiple key attribute definitions in a single request or not. The KMS protocol implementation is responsible for generating the appropriate requests (single/multiple) to the KMS. In certain error situations, the status of each attribute is updated indicating if that attribute was successfully deleted or not.

The `KeyAttributesCount` and `KeyAttributes` parameters are used to specify the key attributes data to be deleted on the KMS. Any number of attributes may be deleted in a single operation,
regardless of whether the KMS supports multiple key attribute definitions in a single request or not. The KMS protocol implementation is responsible for generating the appropriate requests (single/multiple) to the KMS. In certain error situations, the status of each attribute is updated indicating if that attribute was successfully deleted or not.

The `ClientDataSize` and `ClientData` parameters allow the caller to pass an arbitrary block of data to/from the KMS for uses such as auditing or access control. The KMS protocol implementation does not alter this data block other than to package it for transmission to the KMS. The use of these parameters is optional unless the KMS requires it in order to perform the requested action.

**Status Codes Returned**

The `DeleteKeyAttributes()` function will return a status which indicates the overall status of the request. Note that this may be different from the status reported for individual key attribute requests. Status codes returned for the method are:

<table>
<thead>
<tr>
<th>Status Code</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>EFI_SUCCESS</td>
<td>Successfully deleted all requested key attributes.</td>
</tr>
<tr>
<td>EFI_OUT_OF_RESOURCES</td>
<td>Could not allocate required resources.</td>
</tr>
<tr>
<td>EFI_TIMEOUT</td>
<td>Timed out waiting for device or key server. Check individual key attribute request(s) to see which ones may have been processed.</td>
</tr>
<tr>
<td>EFI_ACCESS_DENIED</td>
<td>Access was denied by the device or the key server; OR a <code>ClientId</code> is required by the server and either none or an invalid id was provided.</td>
</tr>
<tr>
<td>EFI_DEVICE_ERROR</td>
<td>Device or key server error. Check individual key attribute request(s) (i.e., key attribute status for each) to see which ones may have been processed.</td>
</tr>
<tr>
<td>EFI_INVALID_PARAMETER</td>
<td>This is <code>NULL</code>, <code>ClientId</code> is required but it is <code>NULL</code>, <code>KeyAttributesCount</code> is <code>NULL</code>, or <code>KeyAttributes</code> is <code>NULL</code>, or <code>KeyIdentifierSize</code> is <code>NULL</code>, or <code>KeyIdentifier</code> is <code>NULL</code>.</td>
</tr>
<tr>
<td>EFI_NOT_FOUND</td>
<td>The <code>KeyIdentifier</code> could not be found or the attribute could not be found. On return the <code>KeyAttributesCount</code> contains the number of attributes processed. Individual structures will reflect the status of the processing for that structure.</td>
</tr>
<tr>
<td>EFI_UNSUPPORTED</td>
<td>The implementation/KMS does not support this function.</td>
</tr>
</tbody>
</table>
EFI_KMS_PROTOCOL.GetKeyByAttributes()

Summary

Retrieve one or more key that has matched all of the specified key attributes.

Prototype

typedef

EFI_STATUS

(EFIAPI *EFI_KMS_GET_KEY_BY_ATTRIBUTES) (  
  IN   EFI_KMS_PROTOCOL    *This,
  IN   EFI_KMS_CLIENT_INFO *Client,
  IN   UINTN               *KeyAttributeCount,
  IN OUT EFI_KMS_KEY_ATTRIBUTE  *KeyAttributes,
  IN OUT UINTN               *KeyDescriptorCount,
  IN OUT EFI_KMS_KEY_DESCRIPTOR *KeyDescriptors,
  IN OUT UINTN               *ClientDataSize OPTIONAL,
  IN OUT VOID                **ClientData OPTIONAL
);

Parameters

This Pointer to this EFI_KMS_PROTOCOL instance.
Client Pointer to a valid EFI_KMS_CLIENT_INFO structure.
KeyAttributeCount Pointer to a count of the number of key attribute structures that must be matched for each returned key descriptor. On input the count value is one or more. On normal returns, this number will be updated with the number of key attributes successfully processed.
KeyAttributes Pointer to an array of EFI_KMS_KEY_ATTRIBUTE structure to search for. On input, the values for the fields in the structure are completely filled in. On return the KeyAttributeStatus field will reflect the result of the operation for each key attribute request.
KeyDescriptorCount Pointer to a count of the number of key descriptors matched by this operation. On entry, this number will be zero. On return, this number will be updated to the number of key descriptors successfully found.
KeyDescriptors Pointer to an array of EFI_KMS_KEY_DESCRIPTOR structures which describe the keys from the KMS having the KeyAttribute(s) specified. On input, this pointer will be NULL. On output, the array will contain an EFI_KMS_KEY_DESCRIPTOR structure for each key meeting the search criteria. Memory for the array and all KeyValue fields will be allocated with the EfiBootServicesData type and must be freed by the caller when it is no longer needed. Also, the KeyStatus field of each descriptor will reflect the result of the request relative to that key descriptor.
**ClientDataSize**

Pointer to the size, in bytes, of an arbitrary block of data specified by the `ClientData` parameter. This parameter may be **NULL**, in which case the `ClientData` parameter will be ignored and no data will be transferred to or from the KMS. If the parameter is not **NULL**, then `ClientData` must be a valid pointer. If the value pointed to is 0, no data will be transferred to the KMS, but data may be returned by the KMS. For all non-zero values `*ClientData` will be transferred to the KMS, which may also return data to the caller. In all cases, the value upon return to the caller will be the size of the data block returned to the caller, which will be zero if no data is returned from the KMS.

**ClientData**

Pointer to a pointer to an arbitrary block of data of `*ClientDataSize` that is to be passed directly to the KMS if it supports the use of client data. This parameter may be **NULL** if and only if the `ClientDataSize` parameter is also **NULL**. Upon return to the caller, `*ClientData` points to a block of data of `*ClientDataSize` that was returned from the KMS. If the returned value for `*ClientDataSize` is zero, then the returned value for `*ClientData` must be **NULL** and should be ignored by the caller. The KMS protocol consumer is responsible for freeing all valid buffers used for client data regardless of whether they are allocated by the caller for input to the function or by the implementation for output back to the caller.

**Description**

The `GetKeyByAttributes()` function returns the keys found by searches for matching key attribute(s). This function must be supported by every KMS protocol instance that supports the use of key attributes as indicated in the protocol's `KeyAttributesSupported` field.

The `Client` parameter identifies the caller to the key management service. It may be used for auditing or access control. The use of this parameter is optional unless the KMS requires it in order to perform the requested action.

The `KeyAttributesCount` and `KeyAttributes` parameters are used to specify the key attributes data to be searched for on the KMS. Any number of attributes may be searched for in a single operation, regardless of whether the KMS supports multiple key attribute definitions in a single request or not. The KMS protocol implementation is responsible for generating the appropriate requests (single/multiple) to the KMS. In certain error situations, the status of each attribute is updated indicating if that attribute was successfully found or not. If an attribute specifies a wildcard `KeyAttributeInstance` value, then the provider returns all instances of the attribute.

The `KeyDescriptorCount` and `KeyDescriptors` parameters are used to return the `EFI_KMS_KEY_DESCRIPTOR` structures for keys meeting the search criteria. Any number of keys may be returned in a single operation, regardless of whether the KMS supports multiple key definitions in a single request or not. The KMS protocol implementation is responsible for generating the appropriate requests (single/multiple) to the KMS.
The ClientDataSize and ClientData parameters allow the caller to pass an arbitrary block of data to/from the KMS for uses such as auditing or access control. The KMS protocol implementation does not alter this data block other than to package it for transmission to the KMS. The use of these parameters is optional unless the KMS requires it in order to perform the requested action.

### Status Codes Returned

The GetKeyByAttributes() function will return a status which indicates the overall status of the request. Note that this may be different from the status reported for individual keys.

<table>
<thead>
<tr>
<th>Status Code</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>EFI_SUCCESS</td>
<td>Successfully retrieved all requested keys.</td>
</tr>
<tr>
<td>EFI_OUT_OF_RESOURCES</td>
<td>Could not allocate required resources.</td>
</tr>
<tr>
<td>EFI_TIMEOUT</td>
<td>Timed out waiting for device or key server. Check individual key attribute request(s) to see which ones may have been processed.</td>
</tr>
<tr>
<td>EFI_BUFFER_TOO_SMALL</td>
<td>If multiple keys are associated with the attribute(s), and the KeyValue buffer does not contain enough structures (KeyDescriptorCount) to contain all the key data, then the available structures will be filled and KeyDescriptorCount will be updated to indicate the number of keys which could not be processed.</td>
</tr>
<tr>
<td>EFI_ACCESS_DENIED</td>
<td>Access was denied by the device or the key server; OR a ClientId is required by the server and either none or an invalid id was provided</td>
</tr>
<tr>
<td>EFI_DEVICE_ERROR</td>
<td>Device or key server error. Check individual key attribute request(s) (i.e., key attribute status for each) to see which ones may have been processed.</td>
</tr>
<tr>
<td>EFI_INVALID_PARAMETER</td>
<td>This is NULL, ClientId is required but it is NULL, KeyDescriptorCount is NULL, or KeyDescriptors is NULL or KeyAttributes is NULL, or KeyAttributesCount is NULL.</td>
</tr>
<tr>
<td>EFI_NOT_FOUND</td>
<td>One or more EFI_KMS_KEY_ATTRIBUTE structures could not be processed properly. KeyAttributeCount contains the number of structures which were successfully processed. Individual structures will reflect the status of the processing for that structure.</td>
</tr>
<tr>
<td>EFI_UNSUPPORTED</td>
<td>The implementation/KMS does not support this function</td>
</tr>
</tbody>
</table>

### 37.4 PKCS7 Verify Protocol

**EFI_PKCS7_VERIFY_PROTOCOL**

**Summary**

The EFI_PKCS7_VERIFY_PROTOCOL may be used to verify data signed with PKCS#7 formatted authentication. The PKCS#7 data to be verified must be binary DER encoded. Additional information on the supported ASN.1 formatting is provided below.

Drivers that supply PKCS7 verification function should publish the EFI_PKCS7_VERIFY_PROTOCOL. Drivers wishing to use the EFI_PKCS7_VERIFY_PROTOCOL may get a reference with LocateProtocol().
# EFI_PKCS7_VERIFY_PROTOCOL_GUID

```c
#define EFI_PKCS7_VERIFY_PROTOCOL_GUID \
{ 0x47889fb2, 0xd671, 0x4fab, \
 0xa0, 0xca, 0xdf, 0xe, 0x44, \
 0xdf, 0x70, 0xd6 }
```

## Protocol Interface Structure

```c
typedef struct _EFI_PKCS7_VERIFY_PROTOCOL {
  EFI_PKCS7_VERIFY_BUFFER VerifyBuffer;
  EFI_PKCS7_VERIFY_SIGNATURE VerifySignature;
} EFI_PKCS7_VERIFY_PROTOCOL;
```

### Parameters

- **VerifyBuffer**
  Examine a DER-encoded PKCS7-signed memory buffer with signature containing embedded data content, or buffer with detached signature and separate data content buffer, and verify using supplied signature lists.

- **VerifySignature**
  Examine a DER-encoded PKCS7-signed memory buffer with signature and, using caller-supplied hash value for signed data, verify using supplied signature lists.

## Description

The **EFI_PKCS7_VERIFY_PROTOCOL** is used to verify data signed using PKCS7 structure. PKCS7 is a general-purpose cryptographic standard (see references). The PKCS7 data to be verified must be ASN.1 (DER) encoded. Implementation must support SHA256 as digest algorithm with RSA digest encryption. Support of other hash algorithms is optional. See Table 37-5.

### Table 37-5  Details of Supported Signature Format.

<table>
<thead>
<tr>
<th>Signature Buffer Format Details</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Encoding</td>
<td>Binary DER</td>
</tr>
<tr>
<td>ASN.1 root of Embedded Signed Data</td>
<td>ContentInfo with <strong>SignedData</strong> content type</td>
</tr>
<tr>
<td>ASN.1 root of Detached Signature</td>
<td><strong>SignedData</strong> or ContentInfo with <strong>SignedData</strong> content type</td>
</tr>
<tr>
<td>Embedded Data Type</td>
<td>Typically ‘Data’ (1.2.840.113549.1.7.1) or other defined OID type (however caller should not depend upon specialized OID processing during PKCS validation.)</td>
</tr>
<tr>
<td>Digest (Hash) Algorithm (VerifyBuffer function)</td>
<td>Support of SHA-256 (2.16.840.1.101.3.4.2.1) is required, other algorithms are optional</td>
</tr>
<tr>
<td>Digest Encryption</td>
<td>RSA (1.2.840.113549.1.1.1)</td>
</tr>
<tr>
<td>Certificate validity dates</td>
<td>See <strong>TimeStampDb</strong> description</td>
</tr>
<tr>
<td>Signature authenticatedAttributes</td>
<td>Ignored by function</td>
</tr>
<tr>
<td>Timestamping</td>
<td>See <strong>TimeStampDb</strong> description</td>
</tr>
</tbody>
</table>
References
PKCS7 is defined by RFC2315. For more information see “Links to UEFI-Related Documents” (http://uefi.org/uefi) under the heading “RFC2315 (defines PKCS7).

EFI_PKCS7_VERIFY_PROTOCOL.VerifyBuffer()

Summary
This function processes a buffer containing binary DER-encoded PKCS7 signature. The signed data content may be embedded within the buffer or separated. Function verifies the signature of the content is valid and signing certificate was not revoked and is contained within a list of trusted signers.

Prototype

typedef
    EFI_STATUS
    (EFIAPI *VerifyBuffer)(
        IN EFI_PKCS7_VERIFY_PROTOCOL *This,
        IN VOID *SignedData,
        IN UINTN SignedDataSize,
        IN VOID *InData OPTIONAL,
        IN UINTN InDataSize
        IN EFI_SIGNATURE_LIST **AllowedDb,
        IN EFI_SIGNATURE_LIST **RevokedDb OPTIONAL,
        IN EFI_SIGNATURE_LIST **TimeStampDb OPTIONAL,
        OUT VOID *Content OPTIONAL,
        IN OUT UINTN *ContentSize
    );

Parameters
This
    Pointer to EFI_PKCS7_VERIFY_PROTOCOL instance.
SignedData
    Points to buffer containing ASN.1 DER-encoded PKCS signature.
SignedDataSize
    The size of SignedData buffer in bytes.
InData
    In case of detached signature, InData points to buffer containing the raw message data previously signed and to be verified by function. In case of SignedData containing embedded data, InData must be NULL.
InDataSize
    When InData is used, the size of InData buffer in bytes. When InData is NULL, this parameter must be 0.
AllowedDb
    Pointer to a list of pointers to EFI_SIGNATURE_LIST structures. The list is terminated by a null pointer. The EFI_SIGNATURE_LIST structures contain lists of X.509 certificates of approved signers. See Chapter 27 for definition of EFI_SIGNATURE_LIST. Function recognizes signer certificates of type EFI_CERT_X509_GUID. Any hash certificate in AllowedDb list is ignored by this function. Function returns
success if signer of the buffer is within this list (and not within `RevokedDb`). This parameter is required.

**RevokedDb**
Optional pointer to a list of pointers to `EFI_SIGNATURE_LIST` structures. The list is terminated by a null pointer. List of X.509 certificates of revoked signers and revoked file hashes. Except as noted in description of `TimeStampDb`, signature verification will always fail if the signer of the file or the hash of the data component of the buffer is in `RevokedDb` list. This list is optional and caller may pass Null or pointer to NULL if not required.

**TimeStampDb**
Optional pointer to a list of pointers to `EFI_SIGNATURE_LIST` structures. The list is terminated by a null pointer. This parameter can be used to pass a list of X.509 certificates of trusted time stamp signers. This list is optional and caller may pass Null or pointer to NULL if not required.

**Content**
On input, points to an optional caller-allocated buffer into which the function will copy the content portion of the file after verification succeeds. This parameter is optional and if NULL, no copy of content from file is performed.

**ContentSize**
On input, points to the size in bytes of the optional buffer `Content` previously allocated by caller. On output, if the verification succeeds, the value referenced by `ContentSize` will contain the actual size of the content from signed file. If `ContentSize` indicates the caller-allocated buffer is too small to contain content, an error is returned, and `ContentSize` will be updated with the required size. This parameter must be 0 if `Content` is Null.

**Description**
This function processes the buffer `SignedData` for PKCS7 verification. The data that was signed using PKCS is referred to as the ‘Message’. In the process of creating a signature of the message, a SHA256 or other hash of the message bytes, called the ‘Message Digest’, is encrypted using a private key held in secret by the signer. The encrypted hash and the X.509 public key certificate of the signer are formatted according to the ASN.1 PKCS#7 Schema (See References). For the buffer type with the embedded data, the ASN.1 syntax is also used to wrap the data and combine the message data with the signature structure.

The `SignedData` buffer must be ASN.1 DER-encoded format with structure according to the subset defined in the introduction to this protocol. Both embedded content and detached signature formats are supported. In case of embedded content, `SignedData` contains both the PKCS7 signature structure and the message content that was signed. In the case of detached signature, `SignedData` contains only the signature data and `InData` is used to supply the data to be verified. To pass verification the X.509 public certificate of the signer of the file must be found in `AllowedDb` and not be present in `RevokedDb`. Additionally if `RevokedDb` contains a specific Hash signature that matches the hash calculated for the content, the file will also fail verification. The message content will be copied to the caller-supplied buffer `Content` (when present) with `ContentSize` updated to reflect the total size in bytes of the extracted content.
The `VerifyBuffer()` function performs several steps. First, the buffer containing the user-provided signature is parsed, the content is located and a hash calculated, and the PKCS7 signature of that hash is verified by decrypting the hash calculated at time of signing. Match of current hash with decrypted hash provides indication the structure contained in buffer has not been modified since signing. Next the protocol function attempts to match the signing certificate included within the signed data again the members of an (optional) list of caller-provided revoked certificates (`RevokedDb`). The hash of the data is also compared against any hash items contained in `RevokedDb` list. Next the signing certificate is matched against the caller-provided list of trusted signatures. If the signature is valid, the certificate or hash are not in the revoked list, and the certificate is in the trusted list, the file passes verification.

When `TimeStampDb` list is present this information modifies the processing of revoked certificates found in both `AllowedDb` and `RevokedDb`. When PCKS7 signings that are time-stamped by trusted signer in `TimeStampDb` list, and which time-stamping occurred prior to the time of certificate revocation noted in certificate in `RevokedDb` list, the signing will be allowed and return `EFI_SUCCESS`. `TimeStampDb` parameter is optional and may be NULL or a pointer to NULL when not used. Except in the processing of certificates found in both `AllowedDb` and `RevokedDb`, `TimeStampDb` is not used and time-stamping is not otherwise required for signings verified by certificate only in `AllowedDb`.

**Note:** This method is intended to be suitable to implement Secure Boot image validation, and as such the contents of `AllowedDb`, `RevokedDb`, and `TimeStampDb` must also conform with the requirements of Authorization Process, bullet item 3 (UEFI Image Validation Succeeded).

The verification function can handle both embedded data or detached signature formats. In case of embedded data, the function will optionally extract the original signed data and supply back to caller in caller-supplied buffer. For a detached signature the caller must provide the original message data in buffer pointed to by `InData`. For consistency, when both `InData` and `Content` are provided, the function will copy contents of `InData` to `Content`.

In case where the `ContentSize` indicated by caller is too small to contain the entire content extracted from the file, `EFI_BUFFER_TOO_SMALL` error is returned, and `ContentSize` is updated to reflect the required size.

**Note:** When signing certificate is matched to `AllowedDb` or `RevokedDb` lists, a match can occur against an entry in the list at any level of the chain of X.509 certificates present in the PCKS certificate list. This supports signing with a certificate that chains to one of the certificates in the `AllowedDb` or `RevokedDb` lists.

**Related Definitions**

None
## Status Codes Returned

<table>
<thead>
<tr>
<th>Code</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>EFI_SUCCESS</td>
<td>Content signature was verified against hash of content, the signer’s certificate was not found in RevokedDb, and was found in AllowedDb or if in signer is found in both AllowedDb and RevokedDb, the signing was allowed by reference to TimeStampDb as described above, and no hash matching content hash was found in RevokedDb.</td>
</tr>
<tr>
<td>EFI_SECURITY_VIOLATION</td>
<td>The SignedData buffer was correctly formatted but signer was in RevokedDb or not in AllowedDb. Also returned if matching content hash found in RevokedDb.</td>
</tr>
<tr>
<td>EFI_COMPROMISED_DATA</td>
<td>Calculated hash differs from signed hash.</td>
</tr>
<tr>
<td>EFI_INVALID_PARAMETER</td>
<td>SignedData is NULL or SignedDataSize is zero. Correctly formatted buffer for processing by the function.</td>
</tr>
<tr>
<td>EFI_INCOMPATIBLE_PARAMETER</td>
<td>Content is not NULL and ContentSize is NULL.</td>
</tr>
<tr>
<td>EFI_ABORTED</td>
<td>Unsupported or invalid format in TimeStampDb, RevokedDb or AllowedDb list contents was detected.</td>
</tr>
<tr>
<td>EFI_NOT_FOUND</td>
<td>Content not found because InData is NULL and no content embedded in SignedData.</td>
</tr>
<tr>
<td>EFI_UNSUPPORTED</td>
<td>The SignedData buffer was not correctly formatted for processing by the function.</td>
</tr>
<tr>
<td>EFI_UNSUPPORTED</td>
<td>Signed data embedded in SignedData but InData is not NULL.</td>
</tr>
<tr>
<td>EFI_BUFFER_TOO_SMALL</td>
<td>The size of buffer indicated by ContentSize is too small to hold the content. ContentSize updated to required size.</td>
</tr>
</tbody>
</table>

### EFI_PKCS7_VERIFY_PROTOCOL.VerifySignature()

#### Summary

This function processes a buffer containing binary DER-encoded detached PKCS7 signature. The hash of the signed data content is calculated and passed by the caller. Function verifies the signature of the content is valid and signing certificate was not revoked and is contained within a list of trusted signers.

Note: the current UEFI specification allows for a variety of hashes. In order to be secure, the users of this protocol should loop over each hash to see if the binary signature is authorized.
Prototype

```c
typedef EFI_STATUS
  (EFIAPI *VerifySignature)(
    IN EFI_PKCS7_VERIFY_PROTOCOL *This,
    IN VOID *Signature,
    IN UINTN SignatureSize,
    IN VOID *InHash,
    IN UINTN InHashSize,
    IN EFI_SIGNATURE_LIST **AllowedDb,
    IN EFI_SIGNATURE_LIST **RevokedDb OPTIONAL,
    IN EFI_SIGNATURE_LIST **TimeStampDb OPTIONAL,
  );
```

Parameters

- **This**: Pointer to `EFI_PKCS7_VERIFY_PROTOCOL` instance.
- **Signature**: Points to buffer containing ASN.1 DER-encoded PKCS detached signature.
- **SignatureSize**: The size of `Signature` buffer in bytes.
- **InHash**: `InHash` points to buffer containing the caller calculated hash of the data. This parameter may not be NULL.
- **InHashSize**: The size in bytes of `InHash` buffer.
- **AllowedDb**: Pointer to a list of pointers to `EFI_SIGNATURE_LIST` structures. The list is terminated by a null pointer. The `EFI_SIGNATURE_LIST` structures contain lists of X.509 certificates of approved signers. See Chapter 27 for definition of `EFI_SIGNATURE_LIST`. Function recognizes signer certificates of type `EFI_CERT_X509_GUID`. Any hash certificate in `AllowedDb` list is ignored by this function. Function returns success if signer of the buffer is within this list (and not within `RevokedDb`). This parameter is required.
- **RevokedDb**: Pointer to a list of pointers to `EFI_SIGNATURE_LIST` structures. The list is terminated by a null pointer. List of X.509 certificates of revoked signers and revoked file hashes. Signature verification will always fail if the signer of the file or the hash of the data component of the buffer is in `RevokedDb` list. This parameter is optional and caller may pass Null if not required.
- **TimeStampDb**: Optional pointer to a list of pointers to `EFI_SIGNATURE_LIST` structures. The list is terminated by a null pointer. This parameter can be used to pass a list of X.509 certificates of trusted time stamp counter-signers.
Description
This function processes the buffer Signature for PKCS7 verification using hash of the data calculated and pass by caller in the InHash buffer. The data that was signed using PKCS is referred to as the ‘Message’. In the process of creating a signature of the message, a hash of the message bytes, called the ‘Message Digest’, is encrypted using a private key held in secret by the signer. The encrypted hash and the X.509 public key certificate of the signer are formatted according to the ASN.1 PKCS#7 Schema (See References). Any data embedded within the PKCS structure is ignored by the function. This function does not support extraction of signature from executable file formats. The address of the PKCS Signature block must be located and passed by the called.

The hash size passed in InHashSize must match the size of the signed hash embedded within the PKCS signature structure or an error is returned.

The SignedData buffer must be ASN.1 DER-encoded format with structure according to the subset defined in the introduction to this protocol. Both embedded content and detached signature formats are supported however embedded data is ignored. To pass verification the X.509 public certificate of the signer of the file must be found in AllowedDb and not be present in RevokedDb. Additionally, if RevokedDb contains a specific Hash signature that matches the hash calculated for the content, the file will also fail verification.

When TimeStampDb list is present this information modifies the processing of revoked certificates found in both AllowedDb and RevokedDb. When PKCS7 signings that are time-stamped by trusted signer in TimeStampDb list, and which time-stamping occurred prior to the time of certificate revocation noted in certificate in RevokedDb list, the signing will be allowed and return EFI_SUCCESS. TimeStampDb parameter is optional and may be NULL or a pointer to NULL when not used. Except in the processing of certificates found in both AllowedDb and RevokedDb, TimeStampDb is not used and time-stamping is not otherwise required for signings verified by certificate only in AllowedDb.

The VerifySignature() function performs several steps. First, the buffer containing the user-provided signature is parsed, (any embedded content is ignored), and the PKCS7 signature of hash data is verified by decrypting the hash calculated at time of signing. Match of caller provided hash with decrypted hash provides indication the signed data has not been modified since signing. Next the protocol function attempts to match the signing certificate included within the signed data again the members of an (optional) list of caller-provided revoked certificates (RevokedDb). The hash of the data is also compared against any hash items contained in RevokedDb list. Next the signing certificate is matched against the caller-provided list of trusted signatures. If the signature is valid, the certificate or hash are not in the revoked list, and the certificate is in the trusted list, the file passes verification.

Note: When a signing certificate is matched to AllowedDb or RevokedDb lists, a match can occur against an entry in the list at any level of the chain of X.509 certificates present in the PKCS certificate list. This supports signing with a certificate that chains to one of the certificates in the AllowedDb or RevokedDb lists.

Note: Because this function uses hashes and the specification contains a variety of hash choices, you should be aware that the check against the RevokedDb list will improperly succeed if the signature is revoked using a different hash algorithm. For this reason, you should either cycle through all UEFI supported hashes to see if one is forbidden, or rely on a single hash choice.
only if the UEFI signature authority only signs and revokes with a single hash (at time of writing, this hash choice is SHA256).

Related Definitions
None

Status Codes Returned

<table>
<thead>
<tr>
<th>Status Code</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>EFI_SUCCESS</td>
<td>Signed hash was verified against caller-provided hash of content, the signer’s certificate was not found in RevokedDb, and was found in AllowedDb or if in signer is found in both AllowedDb and RevokedDb, the signing was allowed by reference to TimeStampDb as described above, and no hash matching content hash was found in RevokedDb.</td>
</tr>
<tr>
<td>EFI_SECURITY_VIOLATION</td>
<td>The SignedData buffer was correctly formatted but signer was in RevokedDb or not in AllowedDb. Also returned if matching content hash found in RevokedDb.</td>
</tr>
<tr>
<td>EFI_COMPROMISED_DATA</td>
<td>Caller provided hash differs from signed hash. Or, caller and encrypted hash are different sizes.</td>
</tr>
<tr>
<td>EFI_INVALID_PARAMETER</td>
<td>Signature is NULL or SignatureSize is zero. InHash is NULL or InhashSize is zero. AllowedDb is NULL.</td>
</tr>
<tr>
<td>EFI_ABORTED</td>
<td>Unsupported or invalid format in TimeStampDb, RevokedDb or AllowedDb list contents was detected.</td>
</tr>
<tr>
<td>EFI_UNSUPPORTED</td>
<td>The Signature buffer was not correctly formatted for processing by the function.</td>
</tr>
</tbody>
</table>

37.5 Random Number Generator Protocol

This section defines the Random Number Generator (RNG) protocol. This protocol is used to provide random numbers for use in applications, or entropy for seeding other random number generators. Consumers of the protocol can ensure that drivers implementing the protocol produce RNG values in a well-known manner.

When a Deterministic Random Bit Generator (DRBG) is used on the output of a (raw) entropy source, its security level must be at least 256 bits.

**EFI_RNG_PROTOCOL**

Summary
This protocol provides standard RNG functions. It can be used to provide random bits for use in applications, or entropy for seeding other random number generators.
GUID

```c
#define EFI_RNG_PROTOCOL_GUID \
{ 0x3152bca5, 0xeade, 0x433d,\n  {0x86, 0x2e, 0xc0, 0x1c, 0xdc, 0x29, 0x1f, 0x44}}
```

Protocol Interface Structure

```c
typedef struct _EFI_RNG_PROTOCOL {
   EFI_RNG_GET_INFO    GetInfo;
   EFI_RNG_GET_RNG     GetRNG;
} EFI_RNG_PROTOCOL;
```

Parameters

- **GetInfo**: Returns information about the random number generation implementation.
- **GetRNG**: Returns the next set of random numbers.

Description

This protocol allows retrieval of RNG values from an UEFI driver. The `GetInfo` service returns information about the RNG algorithms the driver supports. The `GetRNG` service creates a RNG value using an (optionally specified) RNG algorithm.

**EFI_RNG_PROTOCOL.GetInfo**

Summary

Returns information about the random number generation implementation.

Prototype

```c
typedef
EFI_STATUS
(EIFIAP *EFI_RNG_GET_INFO) ( 
    IN EFI_RNG_PROTOCOL    *This,
    IN OUT UINTN           *RNGAlgorithmListSize,
    OUT EFI_RNG_ALGORITHM  *RNGAlgorithmList
);
```

Parameters

- **This**: A pointer to the `EFI_RNG_PROTOCOL` instance.
- **RNGAlgorithmListSize**: On input, the size in bytes of `RNGAlgorithmList`. On output with a return code of `EFI_SUCCESS`, the size in bytes of the data returned in `RNGAlgorithmList`.
- **RNGAlgorithmList**: On output with a return code of `EFI_BUFFER_TOO_SMALL`, the size of `RNGAlgorithmList` required to obtain the list.
- **RNGAlgorithmList**: A caller-allocated memory buffer filled by the driver with one `EFI_RNG_ALGORITHM` element for each supported RNG algorithm. The list must not change across multiple calls to
the same driver. The first algorithm in the list is the default
algorithm for the driver.

Description
This function returns information about supported RNG algorithms.
A driver implementing the RNG protocol need not support more than one RNG algorithm, but shall
support a minimum of one RNG algorithm.

Related Definitions

```c
typedef EFI_GUID EFI_RNG_ALGORITHM;
```

Status Codes Returned

<table>
<thead>
<tr>
<th>Status Code</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>EFI_SUCCESS</td>
<td>The RNG algorithm list was returned successfully.</td>
</tr>
<tr>
<td>EFI_UNSUPPORTED</td>
<td>The service is not supported by this driver.</td>
</tr>
<tr>
<td>EFI_DEVICE_ERROR</td>
<td>The list of algorithms could not be retrieved due to a hardware or firmware error.</td>
</tr>
<tr>
<td>EFI_BUFFER_TOO_SMALL</td>
<td>The buffer <code>RNGAlgorithmList</code> is too small to hold the result.</td>
</tr>
</tbody>
</table>

**EFI_RNG_PROTOCOL.GetRNG**

Summary
Produces and returns an RNG value using either the default or specified RNG algorithm.

Prototype

```c
typedef
  EFI_STATUS
  (EFIAPI *EFI_RNG_GET_RNG) (  
    IN EFI_RNG_PROTOCOL *This,
    IN EFI_RNG_ALGORITHM *RNGAlgorithm, OPTIONAL
    IN UINTN RNGValueLength,
    OUT UINT8 *RNGValue
  )
```

Parameters

- **This**
  A pointer to the `EFI_RNG_PROTOCOL` instance.
- **RNGAlgorithm**
  A pointer to the `EFI_RNG_ALGORITHM` that identifies the RNG algorithm to use. May be `NULL` in which case the function will use its default RNG algorithm.
- **RNGValueLength**
  The length in bytes of the memory buffer pointed to by `RNGValue`. The driver shall return exactly this number of bytes.
RNGValue

A caller-allocated memory buffer filled by the driver with the resulting RNG value.

Description

This function fills the RNGValue buffer with random bytes from the specified RNG algorithm. The driver must not reuse random bytes across calls to this function. It is the caller's responsibility to allocate the RNGValue buffer.

Status Codes Returned

<table>
<thead>
<tr>
<th>Status Code</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>EFI_SUCCESS</td>
<td>The RNG value was returned successfully.</td>
</tr>
<tr>
<td>EFI_UNSUPPORTED</td>
<td>The algorithm specified by RNGAlgorithm is not supported by this driver.</td>
</tr>
<tr>
<td>EFI_DEVICE_ERROR</td>
<td>An RNG value could not be retrieved due to a hardware or firmware error.</td>
</tr>
<tr>
<td>EFI_NOT_READY</td>
<td>There is not enough random data available to satisfy the length requested by RNGValueLength.</td>
</tr>
<tr>
<td>EFI_INVALID_PARAMETER</td>
<td>RNGValue is null or RNGValueLength is zero.</td>
</tr>
</tbody>
</table>

37.5.1 EFI RNG Algorithm Definitions

Summary

This sub-section provides EFI_GUID values for a selection of EFI_RNG_PROTOCOL algorithms. The algorithms listed are optional, not meant to be exhaustive and may be augmented by vendors or other industry standards.

The “raw” algorithm, when supported, is intended to provide entropy directly from the source, without it going through some deterministic random bit generator.
Prototype

#define EFI_RNG_ALGORITHM_SP800_90_HASH_256_GUID \
{0xa7af67cb, 0x603b, 0x4d42,\ 
{0xba, 0x21, 0x70, 0xbf, 0xb6, 0x29, 0x3f, 0x96}}

#define EFI_RNG_ALGORITHM_SP800_90_HMAC_256_GUID \
{0xc5149b43, 0xae85, 0x4f53,\ 
{0x99, 0x82, 0xb9, 0x43, 0x35, 0xd3, 0xa9, 0xe7}}

#define EFI_RNG_ALGORITHM_SP800_90_CTR_256_GUID \
{0x44f0de6e, 0x4d8c, 0x4045,\ 
{0xa8, 0xc7, 0x4d, 0xd1, 0x68, 0x85, 0x6b, 0x9e}}

#define EFI_RNG_ALGORITHM_X9_31_3DES_GUID \
{0x63c4785a, 0xca34, 0x4012,\ 
{0xa3, 0xc8, 0x0b, 0x6a, 0x32, 0x4f, 0x55, 0x46}}

#define EFI_RNG_ALGORITHM_X9_31_AES_GUID \
{0xacd03321, 0x777e, 0x4d3d,\ 
{0xb1, 0xc8, 0x20, 0xcf, 0xd8, 0x88, 0x20, 0xc9}}

#define EFI_RNG_ALGORITHM_RAW \
{0xe43176d7, 0xb6e8, 0x4827,\ 
{0xb7, 0x84, 0x7f, 0xfd, 0xc4, 0xb6, 0x85, 0x61}}

37.5.2 RNG References


NIST, “Recommended Random Number Generator Based on ANSI X9.31 Appendix A.2.4 Using the 3-Key Triple DES and AES Algorithms,” January 2005. See “Links to UEFI-Related Documents” (http://uefi.org/uefi) under the heading “Recommended Random Number Generator Based on ANSI X9.31”.

37.6

Smart Card Reader and Smart Card Edge Protocols

The UEFI Smart Card Reader Protocol provides an abstraction for device to provide smart card reader support. This protocol is very close to Part 5 of PC/SC workgroup specifications and provides an API to applications willing to communicate with a smart card or a smart card reader.
37.6.1 Smart Card Reader Protocol

**EFI_SMART_CARD_READER_PROTOCOL Summary**

Smart card aware application invokes this protocol to get access to an inserted smart card in the reader or to the reader itself.

**GUID**

```c
#define EFI_SMART_CARD_READER_PROTOCOL_GUID \
{0x2a4d1adf, 0x21dc, 0x4b81,\ 
{0xa4, 0x2f, 0x8b, 0x8e, 0xe2, 0x38, 0x00, 0x60}}
```

**Protocol Interface Structure**

```c
typedef struct _EFI_SMART_CARD_READER_PROTOCOL {
  EFI_SMART_CARD_READER_CONNECT   SCardConnect;
  EFI_SMART_CARD_READER_DISCONNECT SCardDisconnect;
  EFI_SMART_CARD_READER_STATUS    SCardStatus;
  EFI_SMART_CARD_READER_TRANSMIT   SCardTransmit;
  EFI_SMART_CARD_READER_CONTROL   SCardControl;
  EFI_SMART_CARD_READER_GET_ATTRIB SCardGetAttrib;
} EFI_SMART_CARD_READER_PROTOCOL;
```

**Members**

- **SCardConnect** Requests a connection to the smart card or smart card reader.
- **SCardDisconnect** Closes the previously open connection.
- **SCardStatus** Provides informations on smart card status and reader name.
- **SCardTransmit** Exchanges data with smart card or smart card reader.
- **SCardControl** Gives direct control to the smart card reader.
- **SCardGetAttrib** Retrieves reader characteristics.

**Description**

This protocol allows UEFI applications to communicate and get/set all necessary information to the smart card reader.

**Overview**

This document aims at defining a standard way for UEFI applications to use a smart card. The key points are:

- Provide an API as close as possible to Part 5 of the existing PC/SC interface. See “Links to UEFI-Related Documents” ([http://uefi.org/uefi](http://uefi.org/uefi)) under the heading “PC/SC Workgroup Specifications”.
- Remove any unnecessary complexity of PC/SC implementation in a classic OS:
  - Assume no connection sharing
  - No resource manager
  - Reduced set of APIs
Note that this document only focuses on PC/SC Part 5 (access to smart card/smart card reader from an application). Abstracting the smart card (Parts 6/9) is not the scope of this document.

Main differences with existing PC/SC implementation on Linux/MacOS/Windows:

- There is no resource manager, driver exposes Part 5 instead of Part 3
- It is not possible to share a smart card between UEFI applications/drivers
- Reader enumeration is different:
  - On classic PC/SC, SCardListReaders is used
  - In UEFI, reader list is available via OpenProtocol/ScardStatus calls

**EFI_SMART_CARD_READER_PROTOCOL.SCardConnect()**

**Summary**

This function requests connection to the smart card or the reader, using the appropriate reset type and protocol.

**Prototype**

```c
EFI_STATUS
(EIFIAPI *EFI_SMART_CARD_READER_PROTOCOL_CONNECT) (  
  IN EFI_SMART_CARD_READER_PROTOCOL *This,  
  IN UINT32          AccessMode,  
  IN UINT32          CardAction,  
  IN UINT32          PreferredProtocols,  
  OUT UINT32         *ActiveProtocol
);
```

**Parameters**

- **This** Indicates a pointer to the calling context. Type **EFI_SMART_CARD_READER_PROTOCOL** is defined in the **EFI_SMART_CARD_READER_PROTOCOL** description.
- **AccessMode** See “related definitions” below.
- **CardAction** **SCARD_CA_NORESET, SCARD_CA_COLDRESET** or **SCARD_CA_WARMRESET**.
- **PreferredProtocols** Bitmask of acceptable protocols. See “related definitions” below.
- **ActiveProtocol** A flag that indicates the active protocol. See “related definitions” below.
Related Definitions

    //
    // Codes for access mode
    //
    #define SCARD_AM_READER 0x0001 // Exclusive access to reader
    #define SCARD_AM_CARD 0x0002 // Exclusive access to card
    //
    // Codes for card action
    //
    #define SCARD_CA_NORESET 0x0000 // Don’t reset card
    #define SCARD_CA_COLDRESET 0x0001 // Perform a cold reset
    #define SCARD_CA_WARMRESET 0x0002 // Perform a warm reset
    #define SCARD_CA_UNPOWER 0x0003 // Power off the card
    #define SCARD_CA_EJECT 0x0004 // Eject the card
    //
    // Protocol types
    //
    #define SCARD_PROTOCOL_UNDEFINED 0x0000
    #define SCARD_PROTOCOL_T0 0x0001
    #define SCARD_PROTOCOL_T1 0x0002
    #define SCARD_PROTOCOL_RAW 0x0004

Description

The **SCardConnect** function requests access to the smart card or the reader. Upon success, it is then possible to call **SCardTransmit**.

If **AccessMode** is set to **SCARD_AM_READER**, **PreferredProtocols** must be set to **SCARD_PROTOCOL_UNDEFINED** and **CardAction** to **SCARD_CA_NORESET** else function fails with **EFI_INVALID_PARAMETER**.
Status Codes Returned

<table>
<thead>
<tr>
<th>Code</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>EFI_SUCCESS</td>
<td>The requested command completed successfully.</td>
</tr>
<tr>
<td>EFI_INVALID_PARAMETER</td>
<td>This is NULL.</td>
</tr>
<tr>
<td>EFI_INVALID_PARAMETER</td>
<td>AccessMode is not valid.</td>
</tr>
<tr>
<td>EFI_INVALID_PARAMETER</td>
<td>CardAction is not valid.</td>
</tr>
<tr>
<td>EFI_INVALID_PARAMETER</td>
<td>Invalid combination of AccessMode/CardAction/PreferredProtocols.</td>
</tr>
<tr>
<td>EFI_NOT_READY</td>
<td>A smart card is inserted but failed to return an ATR.</td>
</tr>
<tr>
<td>EFI_UNSUPPORTED</td>
<td>PreferredProtocols does not contain an available protocol to use.</td>
</tr>
<tr>
<td>EFI_NO_MEDIA</td>
<td>AccessMode is set to SCARD_AM_CARD but there is no smart card inserted.</td>
</tr>
<tr>
<td>EFI_ACCESS_DENIED</td>
<td>Access is already locked by a previous SCardConnect call.</td>
</tr>
<tr>
<td>EFI_DEVICE_ERROR</td>
<td>Any other error condition, typically a reader removal.</td>
</tr>
</tbody>
</table>

**EFI_SMART_CARD_READER_PROTOCOL.SCardDisconnect()**

**Summary**

This function releases a connection previously taken by SCardConnect.

**Prototype**

```c
typedef EFI_STATUS
(EFFECTI_API *EFI_SMART_CARD_READER_PROTOCOL_DISCONNECT) (  
  IN EFI_SMART_CARD_READER_PROTOCOL *This,  
  IN UINT32 CardAction
);
```

**Parameters**

- **This**
  Indicates a pointer to the calling context. Type EFI_SMART_CARD_READER_PROTOCOL is defined in the description.

- **CardAction**
  See “related definitions” for CardAction in SCardConnect description.

**Description**

The SCardDisconnect function releases the lock previously taken by SCardConnect. In case the smart card has been removed before this call, this function returns EFI_SUCCESS. If there is no previous call to SCardConnect, this function returns EFI_SUCCESS.
Status Codes Returned

<table>
<thead>
<tr>
<th>Status Code</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>EFI_SUCCESS</td>
<td>The requested command completed successfully.</td>
</tr>
<tr>
<td>EFI_INVALID_PARAMETER</td>
<td>This is NULL.</td>
</tr>
<tr>
<td>EFI_INVALID_PARAMETER</td>
<td>CardAction value is unknown.</td>
</tr>
<tr>
<td>EFI_UNSUPPORTED</td>
<td>Reader does not support Eject card feature (disconnect was not performed).</td>
</tr>
<tr>
<td>EFI_DEVICE_ERROR</td>
<td>Any other error condition, typically a reader removal.</td>
</tr>
</tbody>
</table>

**EFI_SMART_CARD_READER_PROTOCOL.SCardStatus()**

**Summary**

This function retrieves some basic information about the smart card and reader.

**Prototype**

```c
typedef EFI_STATUS
(EFIAPI *EFI_SMART_CARD_READER_PROTOCOL_STATUS) (
    IN EFI_SMART_CARD_READER_PROTOCOL *This,
    OUT CHAR16            *ReaderName OPTIONAL,
    IN OUT UINTN           *ReaderNameLength OPTIONAL,
    OUT UINT32            *State OPTIONAL,
    OUT UINT32            *CardProtocol OPTIONAL,
    OUT UINT8             *Atr OPTIONAL,
    IN OUT UINTN           *AtrLength OPTIONAL
);
```

**Parameters**

- **This**
  Indicates a pointer to the calling context. Type `EFI_SMART_CARD_READER_PROTOCOL` is defined in the `EFI_SMART_CARD_READER_PROTOCOL` description.

- **ReaderName**
  A pointer to a NULL terminated string that will contain the reader name.

- **ReaderNameLength**
  On input, a pointer to the variable that holds the maximal size, in bytes, of `ReaderName`.
  On output, the required size, in bytes, for `ReaderName`.

- **State**
  Current state of the smart card reader. See “related definitions” below.

- **CardProtocol**
  Current protocol used to communicate with the smart card. See “related definitions” in `SCardConnect`.

- **Atr**
  A pointer to retrieve the ATR of the smart card.

- **AtrLength**
  On input, a pointer to hold the maximum size, in bytes, of `Atr` (usually 33).
  On output, the required size, in bytes, for the smart card ATR.
Related Definitions

```c
//
// Codes for state type
//
#define SCARD_UNKNOWN  0x0000 /* state is unknown */
#define SCARD_ABSENT   0x0001 /* Card is absent */
#define SCARD_INACTIVE 0x0002 /* Card is present and not powered*/
#define SCARD_ACTIVE   0x0003 /* Card is present and powered */
```

Description

The `SCardStatus` function retrieves basic reader and card information.

If `ReaderName`, `State`, `CardProtocol` or `Atr` is NULL, the function does not fail but does not fill in such variables.

If `EFI_SUCCESS` is not returned, `ReaderName` and `Atr` contents shall not be considered as valid.

Status Codes Returned

<table>
<thead>
<tr>
<th>Status Code</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>EFI_SUCCESS</td>
<td>The requested command completed successfully.</td>
</tr>
<tr>
<td>EFI_INVALID_PARAMETER</td>
<td><code>This</code> is NULL</td>
</tr>
<tr>
<td>EFI_INVALID_PARAMETER</td>
<td><code>ReaderName</code> is not NULL but <code>ReaderNameLength</code> is NULL</td>
</tr>
<tr>
<td>EFI_INVALID_PARAMETER</td>
<td><code>Atr</code> is not NULL but <code>AtrLength</code> is NULL</td>
</tr>
<tr>
<td>EFI_BUFFER_TOO_SMALL</td>
<td><code>ReaderNameLength</code> is not big enough to hold the reader name.  <code>ReaderNameLength</code> has been updated to the required value.</td>
</tr>
<tr>
<td>EFI_BUFFER_TOO_SMALL</td>
<td><code>AtrLength</code> is not big enough to hold the ATR.  <code>AtrLength</code> has been updated to the required value.</td>
</tr>
<tr>
<td>EFI_DEVICE_ERROR</td>
<td>Any other error condition, typically a reader removal.</td>
</tr>
</tbody>
</table>

`EFI_SMART_CARD_READER_PROTOCOL.SCardTransmit()`

Summary

This function sends a command to the card or reader and returns its response.
Prototype

typedef

#include "stddef.h"

EFI_STATUS

(EFIAPI *EFI_SMART_CARD_READER_PROTOCOL_TRANSMIT) ( IN EFI_SMART_CARD_READER_PROTOCOL *This,

IN UINT8 *CAPDU,

IN UINTN CAPDULength,

OUT UINT8 *RAPDU,

IN OUT UINTN *RAPDULength );

Parameters

This Indicates a pointer to the calling context. Type EFI_SMART_CARD_READER_PROTOCOL is defined in the EFI_SMART_CARD_READER_PROTOCOL description.

CAPDU A pointer to a byte array that contains the Command APDU to send to the smart card or reader.

CAPDULength Command APDU size, in bytes.

RAPDU A pointer to a byte array that will contain the Response APDU.

RAPDULength On input, the maximum size, in bytes, of the Response APDU. On output, the size, in bytes, of the Response APDU.

Description

This function sends a command to the card or reader and returns its response. The protocol to use to communicate with the smart card has been selected through SCardConnect call.

In case RAPDULength indicates a buffer too small to hold the response APDU, the function fails with EFI_BUFFER_TOO_SMALL.
Note: the caller has to call previously SCardConnect to make sure the reader/card is not already accessed by another application or driver.

Status Codes Returned

<table>
<thead>
<tr>
<th>Status Code</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>EFI_SUCCESS</td>
<td>The requested command completed successfully.</td>
</tr>
<tr>
<td>EFI_INVALID_PARAMETER</td>
<td>This is NULL.</td>
</tr>
<tr>
<td>EFI_INVALID_PARAMETER</td>
<td>CAPDU is NULL or CAPDULength is 0.</td>
</tr>
<tr>
<td>EFI_BUFFER_TOO_SMALL</td>
<td>RAPDULength is not big enough to hold the response APDU. RAPDULength has been updated to the required value.</td>
</tr>
<tr>
<td>EFI_NO_MEDIA</td>
<td>There is no card in the reader.</td>
</tr>
<tr>
<td>EFI_NOT_READY</td>
<td>Card is not powered.</td>
</tr>
<tr>
<td>EFI_PROTOCOL_ERROR</td>
<td>A protocol error has occurred.</td>
</tr>
<tr>
<td>EFI_TIMEOUT</td>
<td>The reader did not respond.</td>
</tr>
<tr>
<td>EFI_ACCESS_DENIED</td>
<td>A communication with the reader/card is already pending.</td>
</tr>
<tr>
<td>EFI_DEVICE_ERROR</td>
<td>Any other error condition, typically a reader removal.</td>
</tr>
</tbody>
</table>

**EFI_SMART_CARD_READER_PROTOCOL.SCardControl()**

**Summary**

This function provides direct access to the reader.

**Prototype**

```c
typedef EFI_STATUS
(EFIAPICALL *EFI_SMART_CARD_READER_PROTOCOL.Control) (EFIAPI *This,
IN EFI_SMART_CARD_READER_PROTOCOL *This,
IN UINT32 ControlCode,
IN UINT8 *InBuffer OPTIONAL,
IN UINTN InBufferLength OPTIONAL,
OUT UINT8 *OutBuffer OPTIONAL,
IN OUT UINTN *OutBufferLength OPTIONAL);
```

**Parameters**

- **This** Indicates a pointer to the calling context. Type EFI_SMART_CARD_READER_PROTOCOL is defined in the EFI_SMART_CARD_READER_PROTOCOL description.
- **ControlCode** The control code for the operation to perform. See “related definitions” below.
- **InBuffer** A pointer to the input parameters.
- **InBufferLength** Size, in bytes, of input parameters.
- **OutBuffer** A pointer to the output parameters.
- **OutBufferLength** On input, maximal size, in bytes, to store output parameters.
On output, the size, in bytes, of output parameters.

**Description**

This function gives direct control to send commands to the driver or the reader.

The **ControlCode** to use is vendor dependant; the only standard code defined is the one to get PC/SC part 10 features. See “related definitions” below.

**InBuffer** and **Outbuffer** may be NULL when **ControlCode** operation does not require them.

**Note:** the caller has to call previously **SCardConnect** to make sure the reader/card is not already accessed by another application or driver.

**Related Definitions**

```
#define SCARD_CTL_CODE(code) (0x42000000 + (code))
#define CM_IOCTL_GET_FEATURE_REQUEST SCARD_CTL_CODE(3400)
```

**Status Codes Returned**

<table>
<thead>
<tr>
<th>Code</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>EFI_SUCCESS</td>
<td>The requested command completed successfully.</td>
</tr>
<tr>
<td>EFI_INVALID_PARAMETER</td>
<td>This is NULL.</td>
</tr>
<tr>
<td>EFI_INVALID_PARAMETER</td>
<td><strong>ControlCode</strong> requires input parameters but:</td>
</tr>
<tr>
<td></td>
<td>• <strong>InBuffer</strong> is NULL or InBufferLenth is NULL —or—</td>
</tr>
<tr>
<td></td>
<td>• <strong>InBuffer</strong> is not NULL but InBufferLenth is less than</td>
</tr>
<tr>
<td>EFI_INVALID_PARAMETER</td>
<td><strong>OutBuffer</strong> is not NULL but OutBufferLength is NULL</td>
</tr>
<tr>
<td>EFI_UNSUPPORTED</td>
<td><strong>ControlCode</strong> is not supported.</td>
</tr>
<tr>
<td>EFI_BUFFER_TOO_SMALL</td>
<td><strong>OutBufferLength</strong> is not big enough to hold the output parameters.</td>
</tr>
<tr>
<td></td>
<td><strong>OutBufferLength</strong> has been updated to the required value.</td>
</tr>
<tr>
<td>EFI_NO_MEDIA</td>
<td>There is no card in the reader and the control code specified requires one.</td>
</tr>
<tr>
<td>EFI_NOT_READY</td>
<td><strong>ControlCode</strong> requires a powered card to operate.</td>
</tr>
<tr>
<td>EFI_PROTOCOL_ERROR</td>
<td>A protocol error has occurred.</td>
</tr>
<tr>
<td>EFI_TIMEOUT</td>
<td>The reader did not respond.</td>
</tr>
<tr>
<td>EFI_ACCESS_DENIED</td>
<td>A communication with the reader/card is already pending.</td>
</tr>
<tr>
<td>EFI_DEVICE_ERROR</td>
<td>Any other error condition, typically a reader removal.</td>
</tr>
</tbody>
</table>

**EFI_SMART_CARD_READER_PROTOCOL.SCardGetAttrib()**

**Summary**

*This function retrieves a reader or smart card attribute.*

**Prototype**

```c
typedef
```
EFI_STATUS
(EFIAPI *EFI_SMART_CARD_READER_PROTOCOL_GET_ATTRIB) (  
 IN EFI_SMART_CARD_READER_PROTOCOL *This,  
 IN UINT32 Attrib,  
 OUT UINT8 *OutBuffer,  
 IN OUT UINTN *OutBufferLength  
);

Parameters

This Indicates a pointer to the calling context. Type  
EFI_SMART_CARD_READER_PROTOCOL is defined in the  
EFI_SMART_CARD_READER_PROTOCOL description.

Attrib Identifier for the attribute to retrieve.  
See “related definitions” below. Note that all attributes might  
not be implemented.

OutBuffer A pointer to a buffer that will contain attribute data.

OutBufferLength On input, maximal size, in bytes, to store attribute data.  
On output, the size, in bytes, of attribute data.

Related Definitions

Possibly supported attrib values are listed in the PC/SC Specification, Part 3. See Section Q for  
document access.

Description

The SCardGetAttrib function retrieves an attribute from the reader driver.
37.6.2 Smart Card Edge Protocol

The Smart Card Edge Protocol provides an abstraction for device to provide Smart Card support.

**EFI_SMART_CARD_EDGE_PROTOCOL**

Summary

Smart Card aware application invokes this protocol to get access to an inserted Smart Card in the reader.

**GUID**

```c
#define EFI_SMART_CARD_EDGE_PROTOCOL_GUID \
    { 0xd317f29b, 0xa325, 0x4712,\n        { 0x9b, 0xf1, 0xc6, 0x19, 0x54, 0xdc, 0x19, 0x8c } }
```

**Protocol Interface Structure**

```c
typedef struct _EFI_SMART_CARD_EDGE_PROTOCOL {
    EFI_SMART_CARD_EDGE_GET_CONTEXT GetContext;
    EFI_SMART_CARD_EDGE_CONNECT Connect;
    EFI_SMART_CARD_EDGE_DISCONNECT Disconnect;
    EFI_SMART_CARD_EDGE_GET_CSN GetCsn;
    EFI_SMART_CARD_EDGE_GET_READER_NAME GetReaderName;
    EFI_SMART_CARD_EDGE_VERIFY_PIN VerifyPin;
    EFI_SMART_CARD_EDGE_GET_PIN_REMAINING GetPinRemaining;
    EFI_SMART_CARD_EDGE_GET_DATA GetData;
    EFI_SMART_CARD_EDGE_GET_CREDENTIAL GetCredential;
    EFI_SMART_CARD_EDGE_SIGN_DATA SignData;
    EFI_SMART_CARD_EDGE_DECRYPT_DATA DecryptData;
    EFI_SMART_CARD_EDGE_BUILD_DH_AGREEMENT BuildDHAgreement;
} EFI_SMART_CARD_EDGE_PROTOCOL;
```
Members

- **GetContext**  Request the driver context.
- **Connect**  Request a connection to the Smart Card.
- **Disconnect**  Close a previously open connection.
- **GetCSN**  Get Card Serial Number.
- **GetReaderName**  Get name of Smart Card reader used.
- **VerifyPin**  Verify Smart Card PIN.
- **GetPinRemaining**  Get number of remaining PIN tries.
- **GetData**  Get specific data.
- **GetCredential**  Get credentials the Smart Card holds.
- **SignData**  Sign a data.
- **DecryptData**  Decrypt a data.
- **BuildDHAgreement**  Construct a DH (Diffie Hellman) agreement for key derivation.

Description

This protocol allows UEFI applications to interface with a Smart Card during boot process for authentication or data signing / decryption, especially if the application has to make use of PKI.

Overview

This document aims at defining a standard way for UEFI applications to use a Smart Card in PKI (Public Key Infrastructure) context. The key points are:

- Each Smart Card or set of Smart Card have specific behavior.
- Smart Card applications often interface with PKCS #11 API or other cryptographic interface like CNG.
- During boot process not all the possibility of a cryptographic interface, like PKCS #11, are useful, for example it is neither the moment to perform Smart Card administration or Smart Card provisioning nor to process debit or credit operation with Smart Card.

Consequently this protocol focused on those points:

- Offering standard access to Smart Card functionalities that:
  - Authenticate User
  - Sign data
  - Decrypt data
  - Get certificates
- With an API that is enough close with PKCS#11 API that it could be considered as a brick to build a “tiny PKCS#11”.
- An implementation of the protocol can be dedicated to a specific Smart Card or a specific set of Smart Card.
- An implementation of the protocol shall poll for Smart Card reader attachment and removal.
- An implementation of the protocol shall poll for Smart Card insertion and removal. On insertion the protocol shall check if it supports this Smart Card.
Typically an implementation of this protocol will lean on a Smart Card reader protocol (\texttt{EFI\_SMART\_CARD\_READER\_PROTOCOL}).

\texttt{EFI\_SMART\_CARD\_EDGE\_PROTOCOL.GetContext()}

\textbf{Summary}
This function retrieves the context driver.

\textbf{Prototype}

\begin{verbatim}
typedef EFI_STATUS (EFIAPI *EFI_SMART_CARD_EDGE_GET_CONTEXT) (  
    IN EFI_SMART_CARD_EDGE_PROTOCOL *This,  
    OUT UINTN          *NumberAidSupported,  
    IN OUT UINTN       *AidTableSize OPTIONAL,  
    OUT SMART_CARD_AID *AidTable OPTIONAL,  
    OUT UINTN          *NumberSCPresent,  
    IN OUT UINTN       *CsnTableSize OPTIONAL,  
    OUT SMART_CARD_CSN *CsnTable OPTIONAL,  
    OUT UINT32          *VersionScEdgeProtocol OPTIONAL  
);  
\end{verbatim}

\textbf{Parameters}

\begin{itemize}
  \item \texttt{This} Indicates a pointer to the calling context. Type \texttt{EFI\_SMART\_CARD\_EDGE\_PROTOCOL} is defined in the \texttt{EFI\_SMART\_CARD\_EDGE\_PROTOCOL} description.
  \item \texttt{NumberAidSupported} Number of AIDs this protocol supports.
  \item \texttt{AidTableSize} On input, number of items allocated for the AID table. On output, number of items returned by protocol.
  \item \texttt{AidTable} Table of the AIDs supported by the protocol.
  \item \texttt{NumberSCPresent} Number of currently present Smart Cards that are supported by protocol.
  \item \texttt{CsnTableSize} On input, the number of items the buffer CSN table can contain. On output, the number of items returned by the protocol.
  \item \texttt{CsnTable} Table of the CSN of the Smart Card present and supported by protocol.
  \item \texttt{VersionScEdgeProtocol} \texttt{EFI\_SMART\_CARD\_EDGE\_PROTOCOL} version.
\end{itemize}
Related Definitions

```
//
// Maximum size for a Smart Card AID (Application IDentifier)
//
#define SCARD_AID_MAXSIZE 0x0010
//
// Size of CSN (Card Serial Number)
//
#define SCARD_CSN_SIZE 0x0010
//
//Current specification version 1.00
//
#define SMART_CARD_EDGE_PROTOCOL_VERSION_1 0x00000100
// Parameters type definition
//
typedef UINT8 SMART_CARD_AID[SCARD_AID_MAXSIZE];
typedef UINT8 SMART_CARD_CSN[SCARD_CSN_SIZE];
```

Description

The **GetContext** function returns the context of the protocol, the application identifiers supported by the protocol and the number and the CSN unique identifier of Smart Cards that are present and supported by protocol.

If `AidTableSize`, `AidTable`, `CsnTableSize`, `CsnTable` or `VersionProtocol` is NULL, the function does not fail but does not fill in such variables.

In case `AidTableSize` indicates a buffer too small to hold all the protocol AID table, only the first `AidTableSize` items of the table are returned in `AidTable`.

In case `CsnTableSize` indicates a buffer too small to hold the entire table of Smart Card CSN present, only the first `CsnTableSize` items of the table are returned in `CsnTable`.

**VersionScEdgeProtocol** returns the version of the **EFI_SMART_CARD_EDGE_PROTOCOL** this driver uses. For this protocol specification value is **SMART_CARD_EDGE_PROTOCOL_VERSION_1**.

In case of Smart Card removal the internal CSN list is immediately updated, even if a connection is opened with that Smart Card.
Status Codes Returned

<table>
<thead>
<tr>
<th>Code</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>EFI_SUCCESS</td>
<td>The requested command completed successfully.</td>
</tr>
<tr>
<td>EFI_INVALID_PARAMETER</td>
<td>This is NULL.</td>
</tr>
<tr>
<td>EFI_INVALID_PARAMETER</td>
<td>NumberSCPresent is NULL.</td>
</tr>
</tbody>
</table>

EFI_SMART_CARD_EDGE_PROTOCOL. Connect()

Summary
This function establish a connection with a Smart Card the protocol support.

Prototype

```c
typedef EFI_STATUS (EFIAPI *EFI_SMART_CARD_EDGE_CONNECT)(
    IN EFI_SMART_CARD_EDGE_PROTOCOL *This,
    OUT EFI_HANDLE *SCardHandle,
    IN UINT8 *ScardCsn OPTIONAL,
    OUT UINT8 *ScardAid OPTIONAL
);
```

Parameters

- **This**: Indicates a pointer to the calling context. Type `EFI_SMART_CARD_EDGE_PROTOCOL` is defined in the `EFI_SMART_CARD_EDGE_PROTOCOL` description.
- **SCardHandle**: Handle on Smart Card connection.
- **ScardCsn**: CSN of the Smart Card the connection has to be established.
- **ScardAid**: AID of the Smart Card the connection has been established.

Description

The **Connect** function establishes a connection with a Smart Card.

In case of success the **SCardHandle** can be used.

If the **ScardCsn** is NULL the connection is established with the first Smart Card the protocol finds in its table of Smart Card present and supported. Else it establish context with the Smart Card whose CSN given by **ScardCsn**.

If **ScardAid** is not NULL the function returns the Smart Card AID the protocol supports.

After a successful connect the **SCardHandle** will remain existing even in case Smart Card removed from Smart Card reader, but all function invoking this **SCardHandle** will fail. **SCardHandle** is released only on Disconnect.
Status Codes Returned

<table>
<thead>
<tr>
<th>Status Code</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>EFI_SUCCESS</td>
<td>The requested command completed successfully.</td>
</tr>
<tr>
<td>EFI_INVALID_PARAMETER</td>
<td><code>This</code> is NULL.</td>
</tr>
<tr>
<td>EFI_INVALID_PARAMETER</td>
<td><code>SCardHandle</code> is NULL.</td>
</tr>
<tr>
<td>EFI_NO_MEDIA</td>
<td>No Smart Card supported by protocol is present, Smart Card with CSN <code>ScardCsn</code> or Reader has been removed. A Disconnect should be performed.</td>
</tr>
</tbody>
</table>

**EFI_SMART_CARD_EDGE_PROTOCOL.Disconnect()**

**Summary**
This function releases a connection previously established by Connect.

**Prototype**
```c
typedef EFI_STATUS (EFIAPI *EFI_SMART_CARD_EDGE_DISCONNECT) (  
    IN EFI_SMART_CARD_EDGE_PROTOCOL *This,  
    IN EFI_HANDLE SCardHandle
);
```

**Parameters**
- `This` Indicates a pointer to the calling context. Type `EFI_SMART_CARD_EDGE_PROTOCOL` is defined in the `EFI_SMART_CARD_EDGE_PROTOCOL` description.
- `SCardHandle` Handle on Smart Card connection to release.

**Description**
The Disconnect function releases the connection previously established by a Connect. In case the Smart Card or the Smart Card reader has been removed before this call, this function returns EFI_SUCCESS.
Status Codes Returned

<table>
<thead>
<tr>
<th>Code</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>EFI_SUCCESS</td>
<td>The requested command completed successfully.</td>
</tr>
<tr>
<td>EFI_INVALID_PARAMETER</td>
<td>This is NULL.</td>
</tr>
<tr>
<td>EFI_INVALID_PARAMETER</td>
<td>No connection for SCardHandle value.</td>
</tr>
</tbody>
</table>

**EFI_SMART_CARD_EDGE_PROTOCOL.GetCsn**

**Summary**

This function returns the Smart Card serial number.

**Prototype**

```c
typedef EFI_STATUS
    (EFIAPI *EFI_SMART_CARD_EDGE_GET_CSN) (  
    IN EFI_SMART_CARD_EDGE_PROTOCOL *This,   
    IN EFI_HANDLE SCardHandle,              
    OUT UINT8 Csn[SCARD_CSN_SIZE]           
    );
```

**Parameters**

- **This** Indicates a pointer to the calling context. Type `EFI_SMART_CARD_EDGE_PROTOCOL` is defined in the `EFI_SMART_CARD_EDGE_PROTOCOL` description.
- **SCardHandle** Handle on Smart Card connection.
- **Csn** The Card Serial number, 16 bytes array.

**Description**

The `GetCsn` function returns the 16 bytes Smart Card Serial number.

**Status Codes Returned**

<table>
<thead>
<tr>
<th>Code</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>EFI_SUCCESS</td>
<td>The requested command completed successfully.</td>
</tr>
<tr>
<td>EFI_INVALID_PARAMETER</td>
<td>This is NULL.</td>
</tr>
<tr>
<td>EFI_INVALID_PARAMETER</td>
<td>No connection for SCardHandle value.</td>
</tr>
<tr>
<td>EFI_NO_MEDIA</td>
<td>Smart Card or Reader of SCardHandle connection has been removed. A Disconnect should be performed.</td>
</tr>
</tbody>
</table>
EFI_SMART_CARD_EDGE_PROTOCOL.GetReaderName

Summary
This function returns the name of the Smart Card reader used for this connection.

Prototype

typedef
EFI_STATUS
(EIFIAMI *EFI_SMART_CARD_EDGE_GET_READER_NAME) (
    IN EFI_SMART_CARD_EDGE_PROTOCOL *This,
    IN EFI_HANDLE SCardHandle,
    IN OUT UINTN *ReaderNameLength,
    OUT CHAR16 *ReaderName OPTIONAL
);

Parameters

This Indicates a pointer to the calling context. Type EFI_SMART_CARD_EDGE_PROTOCOL is defined in the EFI_SMART_CARD_EDGE_PROTOCOL description.

SCardHandle Handle on Smart Card connection.

ReaderNameLength On input, a pointer to the variable that holds the maximal size, in bytes, of ReaderName. On output, the required size, in bytes, for ReaderName.

ReaderName A pointer to a NULL terminated string that will contain the reader name.

Description
The GetReaderName function returns the name of the Smart Card reader used for this connection.

Status Codes Returned

<table>
<thead>
<tr>
<th>Status Code</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>EFI_SUCCESS</td>
<td>The requested command completed successfully.</td>
</tr>
<tr>
<td>EFI_INVALID_PARAMETER</td>
<td>This is NULL.</td>
</tr>
<tr>
<td>EFI_INVALID_PARAMETER</td>
<td>No connection for SCardHandle value.</td>
</tr>
<tr>
<td>EFI_INVALID_PARAMETER</td>
<td>ReaderNameLength is NULL.</td>
</tr>
<tr>
<td>EFI_NO_MEDIA</td>
<td>Smart Card or Reader of SCardHandle connection has been removed. A Disconnect should be performed.</td>
</tr>
</tbody>
</table>

EFI_SMART_CARD_EDGE_PROTOCOL.VerifyPin()  

Summary
This function authenticates a Smart Card user by presenting a PIN code.
Prototype

typedef EFI_STATUS
(EFI_API *EFI_SMART_CARD_EDGE_VERIFY_PIN) (  
  IN EFI_SMART_CARD_EDGE_PROTOCOL *This,  
  IN EFI_HANDLE SCardHandle,  
  IN INT32 PinSize,  
  IN UINT8 *PinCode,  
  OUT BOOLEAN *PinResult,  
  OUT UINT32 *RemainingAttempts OPTIONAL  
);

Parameters

This   Indicates a pointer to the calling context. Type EFI_SMART_CARD_EDGE_PROTOCOL is defined in the EFI_SMART_CARD_EDGE_PROTOCOL description.
SCardHandle   Handle on Smart Card connection.
PinSize   PIN code buffer size.
PinCode   PIN code to present to the Smart Card.
PinResult   Result of PIN code presentation to the Smart Card.
   TRUE when Smart Card founds the PIN code correct.
RemainingAttempts   Number of attempts still possible.

Description

The VerifyPin function presents a PIN code to the Smart Card.

If Smart Card found the PIN code correct the user is considered authenticated to current application, and the function returns TRUE.

Negative or null PinSize value rejected if PinCode is not NULL

A NULL PinCode buffer means the application didn’t know the PIN, in that case:

• If PinSize value is negative the caller only wants to know if the current chain of the elements Smart Card Edge protocol, Smart Card Reader protocol and Smart Card Reader supports the Secure Pin Entry PCSC V2 functionality.
• If PinSize value is positive or null the caller ask to perform the verify PIN using the Secure PIN Entry functionality.

In PinCode buffer, the PIN value is always given in plaintext, in case of secure messaging the SMART_CARD_EDGE_PROTOCOL will be in charge of all intermediate treatments to build the correct Smart Card APDU.
Status Codes Returned

<table>
<thead>
<tr>
<th>Code</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>EFI_SUCCESS</td>
<td>The requested command completed successfully.</td>
</tr>
<tr>
<td>EFI_UNSUPPORTED</td>
<td>Pinsize &lt; 0 and Secure PIN Entry functionality not supported.</td>
</tr>
<tr>
<td>EFI_INVALID_PARAMETER</td>
<td>This is NULL.</td>
</tr>
<tr>
<td>EFI_INVALID_PARAMETER</td>
<td>No connection for SCardHandle value.</td>
</tr>
<tr>
<td>EFI_INVALID_PARAMETER</td>
<td>Bad value for PinSize: value not supported by Smart Card or, negative with PinCode not null.</td>
</tr>
<tr>
<td>EFI_INVALID_PARAMETER</td>
<td>PinResult is NULL.</td>
</tr>
<tr>
<td>EFI_NO_MEDIA</td>
<td>Smart Card or Reader of SCardHandle connection has been removed. A Disconnect should be performed.</td>
</tr>
</tbody>
</table>

**EFI_SMART_CARD_EDGE_PROTOCOL.GetPinRemaining()**

**Summary**

This function gives the remaining number of attempts for PIN code presentation.

**Prototype**

```c
typedef EFI_STATUS (EFIAPI *EFI_SMART_CARD_EDGE_GET_PIN_REMAINING) (  
  IN EFI_SMART_CARD_EDGE_PROTOCOL *This,  
  IN EFI_HANDLE SCardHandle,  
  OUT UINT32 RemainingAttempts
);
```

**Parameters**

- **This** Indicates a pointer to the calling context. Type `EFI_SMART_CARD_EDGE_PROTOCOL` is defined in the `EFI_SMART_CARD_EDGE_PROTOCOL` description.
- **SCardHandle** Handle on Smart Card connection.
- **RemainingAttempts** Number of attempts still possible.

**Description**

The number of attempts to present a correct PIN is limited and depends on Smart Card and on PIN. This function will retrieve the number of remaining possible attempts.
Status Codes Returned

<table>
<thead>
<tr>
<th>Status Code</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>EFI_SUCCESS</td>
<td>The requested command completed successfully.</td>
</tr>
<tr>
<td>EFI_INVALID_PARAMETER</td>
<td>This is NULL.</td>
</tr>
<tr>
<td>EFI_INVALID_PARAMETER</td>
<td>No connection for SCardHandle value.</td>
</tr>
<tr>
<td>EFI_INVALID_PARAMETER</td>
<td>RemainingAttempts is NULL.</td>
</tr>
<tr>
<td>EFI_NO_MEDIA</td>
<td>Smart Card or Reader of SCardHandle connection has been removed. A Disconnect should be performed.</td>
</tr>
</tbody>
</table>
Status Codes Returned

<table>
<thead>
<tr>
<th>Code</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>EFI_SUCCESS</td>
<td>The requested command completed successfully.</td>
</tr>
<tr>
<td>EFI_INVALID_PARAMETER</td>
<td>This is NULL.</td>
</tr>
<tr>
<td>EFI_INVALID_PARAMETER</td>
<td>No connection for SCardHandle value.</td>
</tr>
<tr>
<td>EFI_INVALID_PARAMETER</td>
<td>DataId is NULL.</td>
</tr>
<tr>
<td>EFI_INVALID_PARAMETER</td>
<td>DataSize is NULL.</td>
</tr>
<tr>
<td>EFI_INVALID_PARAMETER</td>
<td>Data is NULL, and *DataSize is not zero.</td>
</tr>
<tr>
<td>EFI_NOT_FOUND</td>
<td>DataId unknown for this driver.</td>
</tr>
<tr>
<td>EFI_BUFFER_TOO_SMALL</td>
<td>The size of Data is too small for the specified data and the required size is returned in DataSize.</td>
</tr>
<tr>
<td>EFI_ACCESS_DENIED</td>
<td>Operation not performed, conditions not fulfilled. PIN not verified.</td>
</tr>
<tr>
<td>EFI_NO_MEDIA</td>
<td>Smart Card or Reader of SCardHandle connection has been removed. A Disconnect should be performed.</td>
</tr>
</tbody>
</table>

**EFI_SMART_CARD_EDGE_PROTOCOL.GetCredentials()**

**Summary**

This function retrieve credentials store into the Smart Card.

**Prototype**

```c
typedef
    EFI_STATUS
    (EFIAPI *EFI_SMART_CARD_EDGE_GET_CREDENTIAL) (    
    IN EFI_SMART_CARD_EDGE_PROTOCOL *This,
    IN EFI_HANDLE SCardHandle,
    IN OUT UINTN *CredentialSize,
    OUT UINT8 *CredentialList OPTIONAL
    );
```

**Parameters**

- **This** Indicates a pointer to the calling context. Type EFI_SMART_CARD_EDGE_PROTOCOL is defined in the EFI_SMART_CARD_EDGE_PROTOCOL description.
- **SCardHandle** Handle on Smart Card connection.
- **CredentialSize** On input, in bytes, the size of buffer to store the list of credential. On output, in bytes, the size of buffer required to store the entire list of credentials.
- **CredentialList** List of credentials stored into the Smart Card. A list of TLV (Tag Length Value) elements organized in containers array.
Related Definitions

// Type of data elements in credentials list
#define SC_EDGE_TAG_HEADER 0x0000  \
    // value of tag field for header,
    // the number of containers
#define SC_EDGE_TAG_CERT 0x0001  // value of tag field for certificate
#define SC_EDGE_TAG_KEY_ID 0x0002 // value of tag field for key index
    // associated with certificate
#define SC_EDGE_TAG_KEY_TYPE 0x0003  // value of tag field for key type
#define SC_EDGE_TAG_KEY_SIZE 0x0004 // value of tag field for key size

// Length of L fields of TLV items
#define SC_EDGE_L_SIZE_HEADER 1  // size of L field for header
#define SC_EDGE_L_SIZE_CERT 2  // size of L field for certificate (big endian)
#define SC_EDGE_L_SIZE_KEY_ID 1  // size of L field for key index
#define SC_EDGE_L_SIZE_KEY_TYPE 1 // size of L field for key type
#define SC_EDGE_L_SIZE_KEY_SIZE 2 // size of L field for key size (big endian)

// Some TLV items have a fixed value for L field
#define SC_EDGE_L_VALUE_HEADER 1  // value of L field for header
#define SC_EDGE_L_VALUE_KEY_ID 1 // value of L field for key index
#define SC_EDGE_L_VALUE_KEY_TYPE 1 // value of L field for key type
#define SC_EDGE_L_VALUE_KEY_SIZE 2 // value of L field for key size

// Possible values for key type
#define SC_EDGE_RSA_EXCHANGE 0x01  // RSA decryption
#define SC_EDGE_RSA_SIGNATURE 0x02 // RSA signature
#define SC_EDGE_ECDSA_256 0x03    // ECDSA signature
#define SC_EDGE_ECDSA_384 0x04    // ECDSA signature
#define SC_EDGE_ECDSA_521 0x05    // ECDSA signature
#define SC_EDGE_ECDH_256 0x06     // ECDH agreement
#define SC_EDGE_ECDH_384 0x07     // ECDH agreement
#define SC_EDGE_ECDH_521 0x08     // ECDH agreement

Description

The function returns a series of items in TLV (Tag Length Value) format.
First TLV item is the header item that gives the number of following containers (0x00, 0x01, Nb containers).
All these containers are a series of 4 TLV items:

- The certificate item (0x01, certificate size, certificate)
- The Key identifier item (0x02, 0x01, key index)
- The key type item (0x03, 0x01, key type)
The key size item (0x04, 0x02, key size), key size in number of bits.

Numeric multi-bytes values are on big endian format, most significant byte first:

- The L field value for certificate (2 bytes)
- The L field value for key size (2 bytes)
- The value field for key size (2 bytes)

**Status Codes Returned**

<table>
<thead>
<tr>
<th>Code</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>EFI_SUCCESS</td>
<td>The requested command completed successfully.</td>
</tr>
<tr>
<td>EFI_INVALID_PARAMETER</td>
<td>This is NULL.</td>
</tr>
<tr>
<td>EFI_INVALID_PARAMETER</td>
<td>No connection for SCardHandle value.</td>
</tr>
<tr>
<td>EFI_INVALID_PARAMETER</td>
<td>CredentialSize is NULL.</td>
</tr>
<tr>
<td>EFI_INVALID_PARAMETER</td>
<td>CredentialList is NULL, if CredentialSize is not zero.</td>
</tr>
<tr>
<td>EFI_BUFFER_TOO_SMALL</td>
<td>The size of CredentialList is too small for the specified data and the</td>
</tr>
<tr>
<td></td>
<td>required size is returned in CredentialSize.</td>
</tr>
<tr>
<td>EFI_NO_MEDIA</td>
<td>Smart Card or Reader of SCardHandle connection has been removed. A</td>
</tr>
<tr>
<td></td>
<td>Disconnect should be performed.</td>
</tr>
</tbody>
</table>

**EFI_SMART_CARD_EDGE_PROTOCOL.SignData()**

**Summary**

This function signs an already hashed data with a Smart Card private key.

**Prototype**

```c
typedef EFI_STATUS
    (EFIAPIC *EFI_SMART_CARD_EDGE_SIGN_DATA)(
    IN EFI_SMART_CARD_EDGE_PROTOCOL *This,
    IN EFI_HANDLE SCardHandle,
    IN UINTN KeyId,
    IN UINTN KeyType,
    IN EFI_GUID *HashAlgorithm,
    IN EFI_GUID *PaddingMethod,
    IN UINT8 *HashedData,
    OUT UINT8 *SignatureData
    );
```

**Parameters**

- **This** Indicates a pointer to the calling context. Type EFI_SMART_CARD_EDGE_PROTOCOL is defined in the EFI_SMART_CARD_EDGE_PROTOCOL description.
- **SCardHandle** Handle on Smart Card connection.
KeyId
Identifier of the key container, retrieved in a key index item of credentials.

KeyType
The key type, retrieved in a key type item of credentials.

HashAlgorithm
Hash algorithm used to hash the, one of:
- EFI_HASH_ALGORITHM_SHA1_GUID
- EFI_HASH_ALGORITHM_SHA256_GUID
- EFI_HASH_ALGORITHM_SHA384_GUID
- EFI_HASH_ALGORITHM_SHA512_GUID

PaddingMethod
Padding method used jointly with hash algorithm, one of:
- EFI_PADDING_RSASSA_PKCS1V1P5_GUID
- EFI_PADDING_RSASSA_PSS_GUID

HashedData
Hash of the data to sign. Size is function of the HashAlgorithm.

SignatureData
Resulting signature with private key KeyId. Size is function of the KeyType and key size retrieved in the associated key size item of credentials.

Related Definitions

    //
    // Padding methods GUIDs for signature
    //

    // RSASSA- PKCS#1-V1.5 padding method, for signature
    //
    #define EFI_PADDING_RSASSA_PKCS1V1P5_GUID
    \{0x9317ec24,0x7cb0,0x4d0e,\{0x8b,0x32,0x2e,0xd9,0x20,0x9c,0xd8,0xaf\}\}

    // RSASSA-PSS padding method, for signature
    //
    #define EFI_PADDING_RSASSA_PSS_GUID
    \{0x7b2349e0,0x522d,0x4f8e,\{0xb9,0x27,0x69,0xd9,0x7c,0x9e,0x79,0x5f\}\}

Description
This function signs data, actually it is the hash of these data that is given to the function.

SignatureData buffer shall be big enough for signature. Signature size is function key size and key type.
Status Codes Returned

<table>
<thead>
<tr>
<th>Status Code</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>EFI_SUCCESS</td>
<td>The requested command completed successfully.</td>
</tr>
<tr>
<td>EFI_INVALID_PARAMETER This is NULL</td>
<td></td>
</tr>
<tr>
<td>EFI_INVALID_PARAMETER No connection for SCardHandle value.</td>
<td></td>
</tr>
<tr>
<td>EFI_INVALID_PARAMETER KeyId is not valid.</td>
<td></td>
</tr>
<tr>
<td>EFI_INVALID_PARAMETER KeyType is not valid or not corresponding to KeyId.</td>
<td></td>
</tr>
<tr>
<td>EFI_INVALID_PARAMETER HashAlgorithm is NULL.</td>
<td></td>
</tr>
<tr>
<td>EFI_INVALID_PARAMETER HashAlgorithm is not valid.</td>
<td></td>
</tr>
<tr>
<td>EFI_INVALID_PARAMETER PaddingMethod is NULL.</td>
<td></td>
</tr>
<tr>
<td>EFI_INVALID_PARAMETER PaddingMethod is not valid.</td>
<td></td>
</tr>
<tr>
<td>EFI_INVALID_PARAMETER HashedData is NULL.</td>
<td></td>
</tr>
<tr>
<td>EFI_INVALID_PARAMETER SignatureData is NULL.</td>
<td></td>
</tr>
<tr>
<td>EFI_ACCESS_DENIED Operation not performed, conditions not fulfilled. PIN not verified.</td>
<td>Smart Card or Reader of SCardHandle connection has been removed. A Disconnect should be performed.</td>
</tr>
<tr>
<td>EFI_NO_MEDIA</td>
<td></td>
</tr>
</tbody>
</table>

### EFI_SMART_CARD_EDGE_PROTOCOL.DecryptData()

#### Summary

This function decrypts data with a PKI/RSA Smart Card private key.

#### Prototype

```c
typedef EFI_STATUS
(DECLSPEC(EFI_FriendlyAPI)) EFI_SMART_CARD_EDGE_PROTOCOL_DecryptData(
    IN EFI_SMART_CARD_EDGE_PROTOCOL *This,
    IN EFI_HANDLE SCardHandle,
    IN UINTN KeyId,
    IN EFI_GUID *HashAlgorithm,
    IN EFI_GUID *PaddingMethod,
    IN UINTN EncryptedSize,
    IN UINT8 *EncryptedData,
    IN OUT UINTN *PlaintextSize,
    OUT UINT8 *PlaintextData
);
```

#### Parameters

- **This**: Indicates a pointer to the calling context. Type EFI_SMART_CARD_EDGE_PROTOCOL is defined in the EFI_SMART_CARD_EDGE_PROTOCOL description.
- **SCardHandle**: Handle on Smart Card connection.
KeyId
Identifier of the key container, retrieved in a key index item of credentials.

HashAlgorithm
Hash algorithm used to hash the, one of:
- EFI_HASH_ALGORITHM_SHA1_GUID
- EFI_HASH_ALGORITHM_SHA256_GUID
- EFI_HASH_ALGORITHM_SHA384_GUID
- EFI_HASH_ALGORITHM_SHA512_GUID

PaddingMethod
Padding method used jointly with hash algorithm, one of:
- EFI_PADDING_NONE_GUID
- EFI_PADDING_RSAES_PKCS1V1P5_GUID
- EFI_PADDING_RSAES_OAEP_GUID

EncryptedSize
Size of data to decrypt

EncryptedData
Data to decrypt

PlaintextSize
On input, in bytes, the size of buffer to store the decrypted data. On output, in bytes, the size of buffer required to store the decrypted data.

PlaintextData
Buffer for decrypted data, padding removed.

Related Definitions

//
// Padding methods GUIDs for decryption
//

// No padding, for decryption
//
#define EFI_PADDING_NONE_GUID \
{0x3629ddb1,0x228c,0x452e,\ 
{0xb6,0x16,0x09,0xed,0x31,0x6a,0x97,0x00}}

// RSAES-PKCS#1-V1.5 padding, for decryption
//
#define EFI_PADDING_RSAES_PKCS1V1P5_GUID \
{0xe1c1d0a9,0x40b1,0x4632,\ 
{0xbd,0xcc,0xd9,0xd6,0xe5,0x29,0x56,0x31}}

// RSAES-OAEP padding, for decryption
//
#define EFI_PADDING_RSAES_OAEP_GUID \
{0xc1e63ac4,0xd0cf,0x4ce6,\ 
{0x83,0x5b,0xee,0xd0,0xe6,0xa8,0xa4,0x5b}}
Description

The function decrypts some PKI / RSA encrypted data with private key securely stored into the Smart Card.

The **KeyId** must reference a key of type **SC_EDGE_RSA_EXCHANGE**.

Status Codes Returned

<table>
<thead>
<tr>
<th>Status Code</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>EFI_SUCCESS</strong></td>
<td>The requested command completed successfully.</td>
</tr>
<tr>
<td><strong>EFI_INVALID_PARAMETER</strong></td>
<td>This is NULL.</td>
</tr>
<tr>
<td><strong>EFI_INVALID_PARAMETER</strong></td>
<td>No connection for <strong>SCardHandle</strong> value.</td>
</tr>
<tr>
<td><strong>EFI_INVALID_PARAMETER</strong></td>
<td><strong>KeyId</strong> is not valid or associated key not of type <strong>SC_EDGE_RSA_EXCHANGE</strong></td>
</tr>
<tr>
<td><strong>EFI_INVALID_PARAMETER</strong></td>
<td><strong>HashAlgorithm</strong> is NULL.</td>
</tr>
<tr>
<td><strong>EFI_INVALID_PARAMETER</strong></td>
<td><strong>PaddingMethod</strong> is NULL.</td>
</tr>
<tr>
<td><strong>EFI_INVALID_PARAMETER</strong></td>
<td><strong>PaddingMethod</strong> is not valid.</td>
</tr>
<tr>
<td><strong>EFI_INVALID_PARAMETER</strong></td>
<td><strong>EncryptedSize</strong> is 0.</td>
</tr>
<tr>
<td><strong>EFI_INVALID_PARAMETER</strong></td>
<td><strong>EncryptedData</strong> is NULL.</td>
</tr>
<tr>
<td><strong>EFI_INVALID_PARAMETER</strong></td>
<td><strong>PlaintextSize</strong> is NULL.</td>
</tr>
<tr>
<td><strong>EFI_INVALID_PARAMETER</strong></td>
<td><strong>PlaintextData</strong> is NULL.</td>
</tr>
<tr>
<td><strong>EFI_ACCESS_DENIED</strong></td>
<td>Operation not performed, conditions not fulfilled. PIN not verified.</td>
</tr>
<tr>
<td><strong>EFI_BUFFER_TOO_SMALL</strong></td>
<td><strong>PlaintextSize</strong> is too small for the plaintext data and the required size is returned in <strong>PlaintextSize</strong>.</td>
</tr>
<tr>
<td><strong>EFI_NO_MEDIA</strong></td>
<td>Smart Card or Reader of <strong>SCardHandle</strong> connection has been removed. A <strong>Disconnect</strong> should be performed.</td>
</tr>
</tbody>
</table>

**EFI_SMART_CARD_EDGE_PROTOCOL.BuildDHAgreement()**

Summary

This function performs a secret Diffie Hellman agreement calculation that would be used to derive a symmetric encryption / decryption key.

Prototype

```c
typedef
EFI_STATUS
(EIFIAPI *EFI_SMART_CARD_EDGE_BUILD_DH_AGREEMENT) (  
    IN EFI_SMART_CARD_EDGE_PROTOCOL *This,
    IN EFI_HANDLE ScardHandle,
    IN UINTN KeyId,
    IN UINT8 *dataQx,
    IN UINT8 *dataQy,
    OUT UINT8 *DHAgreement
) ;
```
Parameters

This Indicates a pointer to the calling context. Type 
EFI_SMART_CARD_EDGE_PROTOCOL is defined in the 
EFI_SMART_CARD_EDGE_PROTOCOL description.

SCardHandle Handle on Smart Card connection.

KeyId Identifier of the key container, retrieved in a key index item of 
credentials.

dataQx Public key x coordinate. Size is the same as key size for KeyId. 
Stored in big endian format.

dataQy Public key y coordinate. Size is the same as key size for KeyId. 
Stored in big endian format.

DHAgreement Buffer for DH agreement computed. Size must be bigger or 
equal to key size for KeyId.

Description

The function compute a DH agreement that should be diversified to generate a symmetric key to 
proceed encryption or decryption.

The application and the Smart Card shall agree on the diversification process.

The KeyId must reference a key of one of the types: SC_EDGE_ECDH_256, SC_EDGE_ECDH_384 
or SC_EDGE_ECDH_521.

Status Codes Returned

<table>
<thead>
<tr>
<th>Code</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>EFI_SUCCESS</td>
<td>The requested command completed successfully.</td>
</tr>
<tr>
<td>EFI_INVALID_PARAMETER</td>
<td>This is NULL.</td>
</tr>
<tr>
<td>EFI_INVALID_PARAMETER</td>
<td>No connection for SCardHandle value.</td>
</tr>
<tr>
<td>EFI_INVALID_PARAMETER</td>
<td>KeyId is not valid.</td>
</tr>
<tr>
<td>EFI_INVALID_PARAMETER</td>
<td>dataQx is NULL.</td>
</tr>
<tr>
<td>EFI_INVALID_PARAMETER</td>
<td>dataQy is NULL.</td>
</tr>
<tr>
<td>EFI_INVALID_PARAMETER</td>
<td>DHAgreement is NULL.</td>
</tr>
<tr>
<td>EFI_ACCESS_DENIED</td>
<td>Operation not performed, conditions not fulfilled. PIN not verified.</td>
</tr>
<tr>
<td>EFI_NO_MEDIA</td>
<td>Smart Card or Reader of SCardHandle connection has been removed. A Disconnect should be performed.</td>
</tr>
</tbody>
</table>
38 - Miscellaneous Protocols

38.1 EFI Timestamp Protocol

EFI_TIMESTAMP_PROTOCOL

Summary
The Timestamp protocol provides a platform independent interface for retrieving a high resolution timestamp counter.

GUID

```c
#define EFI_TIMESTAMP_PROTOCOL_GUID \
{ 0xafbfde41, 0x2e6e, 0x4262,\ 
{ 0xba, 0x65, 0x62, 0xb9, 0x23, 0x6e, 0x54, 0x95 } }
```

Protocol Interface Structure

```c
typedef struct _ EFI_TIMESTAMP_PROTOCOL {
    TIMESTAMP_GET GetTimestamp;
    TIMESTAMP_GET_PROPERTIES GetProperties;
} EFI_TIMESTAMP_PROTOCOL;
```

EFI_TIMESTAMP_PROTOCOL.GetTimestamp()

Summary
Retrieves the current timestamp counter value.

Prototype

```c
typedef UINT64
(EIFIAPI *TIMESTAMP_GET) (VOID);
```

Description
Retrieves the current value of a 64-bit free running timestamp counter.

The counter shall count up in proportion to the amount of time that has passed. The counter value will always roll over to zero. The properties of the counter can be retrieved from GetProperties().

The caller should be prepared for the function to return the same value twice across successive calls. The counter value will not go backwards other than when wrapping, as defined by EndValue in GetProperties().

The frequency of the returned timestamp counter value must remain constant. Power management operations that affect clocking must not change the returned counter frequency. The quantization of counter value updates may vary as long as the value reflecting time passed remains consistent.
Return Value
The current value of the free running timestamp counter.

EFI_TIMESTAMP_PROTOCOL.GetProperties ()

Summary
Obtains timestamp counter properties including frequency and value limits.

Prototype
```c
typedef
EFI_STATUS
(EFIAPI *TIMESTAMP_GET_PROPERTIES) (
    OUT   EFI_TIMESTAMP_PROPERTIES *Properties
);
```

Parameters
- **Properties**
  The properties of the timestamp counter. See "Related Definitions" below.

Description
Retrieves the timestamp counter properties structure.

Related Definitions
```c
typedef struct {
    UINT64   Frequency;
    UINT64   EndValue;
} EFI_TIMESTAMP_PROPERTIES;
```

- **Frequency**
  The frequency of the timestamp counter in Hz.

- **EndValue**
  The value that the timestamp counter ends with immediately before it rolls over. For example, a 64-bit free running counter would have an EndValue of 0xFFFFFFFFFFFFFFFF. A 24-bit free running counter would have an EndValue of 0xFFFFFFFF.
Status Codes Returned

<table>
<thead>
<tr>
<th>Code</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>EFI_SUCCESS</td>
<td>The properties were successfully retrieved.</td>
</tr>
<tr>
<td>EFI_DEVICE_ERROR</td>
<td>An error occurred trying to retrieve the properties of the timestamp counter subsystem. Properties is not updated.</td>
</tr>
<tr>
<td>EFI_INVALID_PARAMETER</td>
<td>Properties is NULL.</td>
</tr>
</tbody>
</table>

38.2 Reset Notification Protocol

EFI_RESET_NOTIFICATION_PROTOCOL

Summary
This protocol provides services to register for a notification when ResetSystem is called.

GUID

```c
#define EFI_RESET_NOTIFICATION_PROTOCOL_GUID \
{ 0x9da34ae0, 0xeaf9, 0x4bbf, \n  { 0x8e, 0xc3, 0xfd, 0x60, 0x22, 0x6c, 0x44, 0xbe } }
```

Protocol Interface Structure

```c
typedef struct _EFI_RESET_NOTIFICATION_PROTOCOL {
    EFI_REGISTER_RESET_NOTIFY RegisterResetNotify;
    EFI_UNREGISTER_RESET_NOTIFY UnRegisterResetNotify;
} EFI_RESET_NOTIFICATION_PROTOCOL;
```

Parameters

RegisterResetNotify

Register a notification function to be called when ResetSystem() is called.

UnRegisterResetNotify

Removes a reset notification function that has been previously registered with RegisterResetNotify().

EFI_RESET_NOTIFICATION_PROTOCOL.RegisterResetNotify()

Summary
Register a notification function to be called when ResetSystem() is called.

Prototype

```c
typedef
EFI_STATUS
(EFI_API *EFI_REGISTER_RESET_NOTIFY) ( 
    IN EFI_RESET_NOTIFICATION_PROTOCOL  *This,
    IN EFI_RESET_SYSTEM               *ResetFunction,
);
```
Parameters

This

A pointer to the EFI_RESET_NOTIFICATION_PROTOCOL instance.

ResetFunction

Points to the function to be called when a ResetSystem() is executed.

Description

The RegisterResetNotify() function registers a notification function that is called when ResetSystem() is called and prior to completing the reset of the platform.

The registered functions must not perform a platform reset themselves. These notifications are intended only for the notification of components which may need some special-purpose maintenance prior to the platform resetting.

The list of registered reset notification functions are processed if ResetSystem() is called before ExitBootServices(). The list of registered reset notification functions is ignored if ResetSystem() is called after ExitBootServices().

Status Codes Returned

<table>
<thead>
<tr>
<th>Status Code</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>EFI_SUCCESS</td>
<td>The reset notification function was successfully registered.</td>
</tr>
<tr>
<td>EFI_INVALID_PARAMETER</td>
<td>ResetFunction is NULL.</td>
</tr>
<tr>
<td>EFI_OUT_OF_RESOURCES</td>
<td>There are not enough resources available to register the reset notification function.</td>
</tr>
<tr>
<td>EFI_ALREADY_STARTED</td>
<td>The reset notification function specified by ResetFunction has already been registered.</td>
</tr>
</tbody>
</table>

EFI_RESET_NOTIFICATION_PROTOCOL.UnregisterResetNotify()

Summary

Unregister a notification function.

Prototype

```c
typedef
EFI_STATUS
(EIFIAPI *EFI_UNREGISTER_RESET_NOTIFY) (  
    IN EFI_RESET_NOTIFICATION_PROTOCOL*This,
    IN EFI_RESET_SYSTEM *ResetFunction
);
```
Parameters

This

A pointer to the EFI_RESET_NOTIFICATION_PROTOCOL instance.

ResetFunction

The pointer to the ResetFunction being unregistered.

Description

The UnregisterResetNotify() function removes the previously registered notification using RegisterResetNotify().

Status Codes Returned

<table>
<thead>
<tr>
<th>Status Code</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>EFI_SUCCESS</td>
<td>The reset notification function was unregistered.</td>
</tr>
<tr>
<td>EFI_INVALID_PARAMETER</td>
<td>ResetFunction is NULL.</td>
</tr>
<tr>
<td>EFI_INVALID_PARAMETER</td>
<td>The reset notification function specified by ResetFunction was not previously registered using RegisterResetNotify().</td>
</tr>
</tbody>
</table>
Appendix A - GUID and Time Formats

All EFI GUIDs (Globally Unique Identifiers) have the format described in RFC 4122 and comply with the referenced algorithms for generating GUIDs. It should also be noted that TimeLow, TimeMid, TimeHighAndVersion fields in the EFI are encoded as little endian. The following table defines the format of an EFI GUID (128 bits).

Table A-1 EFI GUID Format

<table>
<thead>
<tr>
<th>Mnemonic</th>
<th>Byte Offset</th>
<th>Byte Length</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>TimeLow</td>
<td>0</td>
<td>4</td>
<td>The low field of the timestamp.</td>
</tr>
<tr>
<td>TimeMid</td>
<td>4</td>
<td>2</td>
<td>The middle field of the timestamp.</td>
</tr>
<tr>
<td>TimeHighAndVersion</td>
<td>6</td>
<td>2</td>
<td>The high field of the timestamp multiplexed with the version number.</td>
</tr>
<tr>
<td>ClockSeqHighAndReserved</td>
<td>8</td>
<td>1</td>
<td>The high field of the clock sequence multiplexed with the variant.</td>
</tr>
<tr>
<td>ClockSeqLow</td>
<td>9</td>
<td>1</td>
<td>The low field of the clock sequence.</td>
</tr>
<tr>
<td>Node</td>
<td>10</td>
<td>6</td>
<td>The spatially unique node identifier. This can be based on any IEEE 802 address obtained from a network card. If no network card exists in the system, a cryptographic-quality random number can be used.</td>
</tr>
</tbody>
</table>

This appendix for GUID defines a 60-bit timestamp format that is used to generate the GUID. All EFI time information is stored in 64-bit structures that contain the following format: The timestamp is a 60-bit value containing a count of 100-nanosecond intervals since 00:00:00.00, 15 October 1582 (the date of Gregorian reform to the Christian calendar). This time value will not roll over until the year 3400 AD. It is assumed that a future version of the EFI specification can deal with the year-3400 issue by extending this format if necessary.

This specification also defines a standard text representation of the GUID. This format is also sometimes called the “registry format”. It consists of 36 characters, as follows:

```
aabbccdd-eeff-gghh-iijj-kkllmnnopp
```

The pairs aa – pp are two characters in the range ‘0’-'9', ‘a’-'f' or ‘A’-'F', with each pair representing a single byte hexadecimal value.

The following table describes the relationship between the text representation and a 16-byte buffer, the structure defined in Table A-1 and the EFI_GUID structure.

Table A-2 Text representation relationships

<table>
<thead>
<tr>
<th>String</th>
<th>Offset In Buffer</th>
<th>Relationship To Table A-1</th>
<th>Relationship To EFI_GUID</th>
</tr>
</thead>
<tbody>
<tr>
<td>bb</td>
<td>2</td>
<td>TimeLow[16:23]</td>
<td>Data1[16:23]</td>
</tr>
<tr>
<td>cc</td>
<td>1</td>
<td>TimeLow[8:15]</td>
<td>Data1[8:15]</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>---</td>
<td>---</td>
<td>---</td>
<td></td>
</tr>
<tr>
<td>dd</td>
<td>0</td>
<td>TimeLow[0:7]</td>
<td>Data1[0:7]</td>
</tr>
<tr>
<td>ee</td>
<td>5</td>
<td>TimeMid[8:15]</td>
<td>Data2[8:15]</td>
</tr>
<tr>
<td>ff</td>
<td>4</td>
<td>TimeMid[0:7]</td>
<td>Data2[0:7]</td>
</tr>
<tr>
<td>hh</td>
<td>6</td>
<td>TimeHighAndVersion[0:7]</td>
<td>Data3[0:7]</td>
</tr>
<tr>
<td>ii</td>
<td>8</td>
<td>ClockSeqHighAndReserved[0:7]</td>
<td>Data4[0:7]</td>
</tr>
<tr>
<td>jj</td>
<td>9</td>
<td>ClockSeqLow[0:7]</td>
<td>Data4[8:15]</td>
</tr>
<tr>
<td>kk</td>
<td>10</td>
<td>Node[0:7]</td>
<td>Data4[16:23]</td>
</tr>
<tr>
<td>mm</td>
<td>12</td>
<td>Node[16:23]</td>
<td>Data4[32:39]</td>
</tr>
<tr>
<td>pp</td>
<td>15</td>
<td>Node[40:47]</td>
<td>Data4[56:63]</td>
</tr>
</tbody>
</table>
Appendix B - Console

The EFI console was designed to allow input from a wide variety of devices. This appendix provides examples of the mapping of keyboard input from various types of devices to EFI scan codes. While representative of common console devices in use today, it is not intended to be a comprehensive list. EFI application programmers can use this table to identify the EFI Scan Code generated by a specific key press. The description of the example device input data that generates a EFI Scan Code may be useful to EFI driver writers, as well as showing the limitations on which EFI Scan codes can be generated by different types of console input devices.

The EFI console was designed so that it could map to common console devices. This appendix explains how an EFI console could map to a VGA with PC AT 101/102, PC ANSI, or ANSI X3.64 consoles.

B.1 EFI_SIMPLE_TEXT_INPUT_PROTOCOL and EFI_SIMPLE_TEXT_INPUT_EX_PROTOCOL

Table B-1 and Table B-2 give examples of how input from a set of common input devices is mapped to EFI scan codes. Terminals and terminal emulators generally report function and editing keys as escape or control sequences. These sequences are formed by a control character followed by one or more additional graphic characters that indicate what the sequence means. ANSI X3.64 terminals generally require an ANSI parser to determine how to interpret a sequence and how to determine that the sequence is complete. These terminals can generate sequences using either 8-bit controls or 7-bit control sequences. Older terminal types, such as the VT100+ have a simpler set of sequences that can be interpreted using simple case statements. These terminals usually generate only 7-bit data, and 7-bit control sequences.

In the tables below, the CSI character is the 8-bit control character 0x9B, and is equivalent to the 7-bit control sequence "ESC [" (the 0x1B control ESC followed by the left bracket character 0x5B). The sequences are shown with spaces for readability, but do not contain the space character.

The VT100+ column represents a common class of terminal emulation that is a superset of the Digital Equipment Corporation (DEC) VT100 terminal. This includes VT-UTF8 (Hyperterm) and PC_ANSI terminal types. The ANSI X3.64 column shows the sequences generated by the DEC VT200 through VT500 terminals, which are an ANSI X3.64 / ISO 6429 compliant.

The USB HID and AT 101/102 columns show the scan codes generated by two common directly attached keyboards. These keyboards are generally used in combination with a VGA text display to form a "VGA Console".

In the table below, the cells with N/A contained in them are simply intended to reflect that the key may be defined for that terminal or keyboard, but there is no industry standard or consistent mapping for the key. Some input devices might not implement all of these keys.
### Table B-1 EFI Scan Codes for EFI_SIMPLE_TEXT_INPUT_PROTOCOL

<table>
<thead>
<tr>
<th>EFI Scan Code</th>
<th>Description</th>
<th>ANSI X3.64 / DEC VT200-500 (8-bit mode)</th>
<th>VT100+ (7-bit mode)</th>
<th>USB Keyboard HID Values</th>
<th>AT 101/102 Keyboard Scan Codes</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x00</td>
<td>Null scan code</td>
<td>N/A</td>
<td>N/A</td>
<td>0x00</td>
<td>N/A</td>
</tr>
<tr>
<td>0x01</td>
<td>UP ARROW</td>
<td>CSI A</td>
<td>ESC [ A</td>
<td>0x52</td>
<td>0xe0, 0x48</td>
</tr>
<tr>
<td>0x02</td>
<td>DOWN ARROW</td>
<td>CSI B</td>
<td>ESC [ B</td>
<td>0x51</td>
<td>0xe0, 0x50</td>
</tr>
<tr>
<td>0x03</td>
<td>RIGHT ARROW</td>
<td>CSI C</td>
<td>ESC [ C</td>
<td>0x4F</td>
<td>0xe0, 0x4d</td>
</tr>
<tr>
<td>0x04</td>
<td>LEFT ARROW</td>
<td>CSI D</td>
<td>ESC [ D</td>
<td>0x50</td>
<td>0xe0, 0x4b</td>
</tr>
<tr>
<td>0x05</td>
<td>Home</td>
<td>CSI 1 ~</td>
<td>ESC h</td>
<td>0x4A</td>
<td>0xe0, 0x47</td>
</tr>
<tr>
<td>0x06</td>
<td>End</td>
<td>CSI 4 ~</td>
<td>ESC k</td>
<td>0x4D</td>
<td>0xe0, 0x4f</td>
</tr>
<tr>
<td>0x07</td>
<td>Insert</td>
<td>CSI 2 ~</td>
<td>ESC +</td>
<td>0x49</td>
<td>0xe0, 0x52</td>
</tr>
<tr>
<td>0x08</td>
<td>Delete</td>
<td>CSI 3 ~</td>
<td>ESC -</td>
<td>0x4C</td>
<td>0xe0, 0x53</td>
</tr>
<tr>
<td>0x09</td>
<td>Page Up</td>
<td>CSI 5 ~</td>
<td>ESC ?</td>
<td>0x4B</td>
<td>0xe0, 0x49</td>
</tr>
<tr>
<td>0x0a</td>
<td>Page Down</td>
<td>CSI 6 ~</td>
<td>ESC /</td>
<td>0x4E</td>
<td>0xe0, 0x51</td>
</tr>
<tr>
<td>0x0b</td>
<td>Function 1</td>
<td>CSI 1 1 ~</td>
<td>ESC 1</td>
<td>0x3A</td>
<td>0x3b</td>
</tr>
<tr>
<td>0x0c</td>
<td>Function 2</td>
<td>CSI 1 2 ~</td>
<td>ESC 2</td>
<td>0x3B</td>
<td>0x3c</td>
</tr>
<tr>
<td>0x0d</td>
<td>Function 3</td>
<td>CSI 1 3 ~</td>
<td>ESC 3</td>
<td>0x3C</td>
<td>0x3d</td>
</tr>
<tr>
<td>0x0e</td>
<td>Function 4</td>
<td>CSI 1 4 ~</td>
<td>ESC 4</td>
<td>0x3D</td>
<td>0x3e</td>
</tr>
<tr>
<td>0x0f</td>
<td>Function 5</td>
<td>CSI 1 5 ~</td>
<td>ESC 5</td>
<td>0x3E</td>
<td>0x3f</td>
</tr>
<tr>
<td>0x10</td>
<td>Function 6</td>
<td>CSI 1 7 ~</td>
<td>ESC 6</td>
<td>0x3F</td>
<td>0x40</td>
</tr>
<tr>
<td>0x11</td>
<td>Function 7</td>
<td>CSI 1 8 ~</td>
<td>ESC 7</td>
<td>0x40</td>
<td>0x41</td>
</tr>
<tr>
<td>0x12</td>
<td>Function 8</td>
<td>CSI 1 9 ~</td>
<td>ESC 8</td>
<td>0x41</td>
<td>0x42</td>
</tr>
<tr>
<td>0x13</td>
<td>Function 9</td>
<td>CSI 2 0 ~</td>
<td>ESC 9</td>
<td>0x42</td>
<td>0x43</td>
</tr>
<tr>
<td>0x14</td>
<td>Function 10</td>
<td>CSI 2 1 ~</td>
<td>ESC 0</td>
<td>0x43</td>
<td>0x44</td>
</tr>
<tr>
<td>0x17</td>
<td>Escape</td>
<td>ESC</td>
<td>ESC</td>
<td>0x29</td>
<td>0x01</td>
</tr>
</tbody>
</table>
### Table B-2: EFI Scan Codes for EFI_SIMPLE_TEXT_INPUT_EX_PROTOCOL

<table>
<thead>
<tr>
<th>EFI Scan Code</th>
<th>Description</th>
<th>ANSI X3.64 / DEC VT200-500 (8-bit mode)</th>
<th>VT100+ (7-bit mode)</th>
<th>USB Keyboard HID Values</th>
<th>AT 101/102 Keyboard Scan Codes</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x15</td>
<td>Function 11</td>
<td>CSI 2 3 ~</td>
<td>ESC !</td>
<td>0x44</td>
<td>0x57</td>
</tr>
<tr>
<td>0x16</td>
<td>Function 12</td>
<td>CSI 2 4 ~</td>
<td>ESC @</td>
<td>0x45</td>
<td>0x58</td>
</tr>
<tr>
<td>0x48</td>
<td>Pause</td>
<td>N/A</td>
<td>N/A</td>
<td>0x48</td>
<td>0xe1, 0x1d, 0x45</td>
</tr>
<tr>
<td>0x68</td>
<td>Function 13</td>
<td>CSI 2 5 ~</td>
<td>N/A</td>
<td>0x68</td>
<td>N/A</td>
</tr>
<tr>
<td>0x69</td>
<td>Function 14</td>
<td>CSI 2 6 ~</td>
<td>N/A</td>
<td>0x69</td>
<td>N/A</td>
</tr>
<tr>
<td>0x6A</td>
<td>Function 15</td>
<td>CSI 2 7 ~</td>
<td>N/A</td>
<td>0x6A</td>
<td>N/A</td>
</tr>
<tr>
<td>0x6B</td>
<td>Function 16</td>
<td>CSI 2 8 ~</td>
<td>N/A</td>
<td>0x6B</td>
<td>N/A</td>
</tr>
<tr>
<td>0x6C</td>
<td>Function 17</td>
<td>CSI 2 9 ~</td>
<td>N/A</td>
<td>0x6C</td>
<td>N/A</td>
</tr>
<tr>
<td>0x6D</td>
<td>Function 18</td>
<td>CSI 3 0 ~</td>
<td>N/A</td>
<td>0x6D</td>
<td>N/A</td>
</tr>
<tr>
<td>0x6E</td>
<td>Function 19</td>
<td>CSI 3 1 ~</td>
<td>N/A</td>
<td>0x6E</td>
<td>N/A</td>
</tr>
<tr>
<td>0x6F</td>
<td>Function 20</td>
<td>CSI 3 2 ~</td>
<td>N/A</td>
<td>0x6F</td>
<td>N/A</td>
</tr>
<tr>
<td>0x70</td>
<td>Function 21</td>
<td>N/A</td>
<td>N/A</td>
<td>0x70</td>
<td>N/A</td>
</tr>
<tr>
<td>0x71</td>
<td>Function 22</td>
<td>N/A</td>
<td>N/A</td>
<td>0x71</td>
<td>N/A</td>
</tr>
<tr>
<td>0x72</td>
<td>Function 23</td>
<td>N/A</td>
<td>N/A</td>
<td>0x72</td>
<td>N/A</td>
</tr>
<tr>
<td>0x73</td>
<td>Function 24</td>
<td>N/A</td>
<td>N/A</td>
<td>0x73</td>
<td>N/A</td>
</tr>
<tr>
<td>0x7F</td>
<td>Mute</td>
<td>N/A</td>
<td>N/A</td>
<td>0x7F</td>
<td>N/A</td>
</tr>
<tr>
<td>0x80</td>
<td>Volume Up</td>
<td>N/A</td>
<td>N/A</td>
<td>0x80</td>
<td>N/A</td>
</tr>
<tr>
<td>0x81</td>
<td>Volume Down</td>
<td>N/A</td>
<td>N/A</td>
<td>0x81</td>
<td>N/A</td>
</tr>
<tr>
<td>0x100</td>
<td>Brightness Up</td>
<td>N/A</td>
<td>N/A</td>
<td>N/A</td>
<td>N/A</td>
</tr>
<tr>
<td>0x101</td>
<td>Brightness Down</td>
<td>N/A</td>
<td>N/A</td>
<td>N/A</td>
<td>N/A</td>
</tr>
<tr>
<td>0x102</td>
<td>Suspend</td>
<td>N/A</td>
<td>N/A</td>
<td>N/A</td>
<td>N/A</td>
</tr>
<tr>
<td>0x103</td>
<td>Hibernate</td>
<td>N/A</td>
<td>N/A</td>
<td>N/A</td>
<td>N/A</td>
</tr>
<tr>
<td>0x104</td>
<td>Toggle Display</td>
<td>N/A</td>
<td>N/A</td>
<td>N/A</td>
<td>N/A</td>
</tr>
<tr>
<td>0x105</td>
<td>Recovery</td>
<td>N/A</td>
<td>N/A</td>
<td>N/A</td>
<td>N/A</td>
</tr>
<tr>
<td>0x106</td>
<td>Eject</td>
<td>N/A</td>
<td>N/A</td>
<td>N/A</td>
<td>N/A</td>
</tr>
<tr>
<td>0x8000-0xFFFF</td>
<td>OEM Reserved</td>
<td>N/A</td>
<td>N/A</td>
<td>N/A</td>
<td>N/A</td>
</tr>
</tbody>
</table>
B.2 EFI_SIMPLE_TEXT_OUTPUT_PROTOCOL

Table B-3 defines how the programmatic methods of the `EFI_SIMPLE_TEXT_OUTPUT_PROTOCOL` could be implemented as PC ANSI or ANSI X3.64 terminals. Detailed descriptions of PC ANSI and ANSI X3.64 escape sequences are as follows. The same type of operations can be supported via a PC AT type INT 10h interface.

Table B-3 Control Sequences to Implement `EFI_SIMPLE_TEXT_INPUT_PROTOCOL`

<table>
<thead>
<tr>
<th>PC ANSI Codes</th>
<th>ANSI X3.64 Codes</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>ESC [ 2 J</td>
<td>CSI 2 J</td>
<td>Clear Display Screen.</td>
</tr>
<tr>
<td>ESC [ 0 m</td>
<td>CSI 0 m</td>
<td>Normal Text.</td>
</tr>
<tr>
<td>ESC [ 1 m</td>
<td>CSI 1 m</td>
<td>Bright Text.</td>
</tr>
<tr>
<td>ESC [ 7 m</td>
<td>CSI 7 m</td>
<td>Reversed Text.</td>
</tr>
<tr>
<td>ESC [ 30 m</td>
<td>CSI 30 m</td>
<td>Black foreground, compliant with ISO Standard 6429.</td>
</tr>
<tr>
<td>ESC [ 31 m</td>
<td>CSI 31 m</td>
<td>Red foreground, compliant with ISO Standard 6429.</td>
</tr>
<tr>
<td>ESC [ 32 m</td>
<td>CSI 32 m</td>
<td>Green foreground, compliant with ISO Standard 6429.</td>
</tr>
<tr>
<td>ESC [ 33 m</td>
<td>CSI 33 m</td>
<td>Yellow foreground, compliant with ISO Standard 6429.</td>
</tr>
<tr>
<td>ESC [ 34 m</td>
<td>CSI 34 m</td>
<td>Blue foreground, compliant with ISO Standard 6429.</td>
</tr>
<tr>
<td>ESC [ 35 m</td>
<td>CSI 35 m</td>
<td>Magenta foreground, compliant with ISO Standard 6429.</td>
</tr>
<tr>
<td>ESC [ 36 m</td>
<td>CSI 36 m</td>
<td>Cyan foreground, compliant with ISO Standard 6429.</td>
</tr>
<tr>
<td>ESC [ 37 m</td>
<td>CSI 37 m</td>
<td>White foreground, compliant with ISO Standard 6429.</td>
</tr>
<tr>
<td>ESC [ 40 m</td>
<td>CSI 40 m</td>
<td>Black background, compliant with ISO Standard 6429.</td>
</tr>
<tr>
<td>ESC [ 41 m</td>
<td>CSI 41 m</td>
<td>Red background, compliant with ISO Standard 6429.</td>
</tr>
<tr>
<td>ESC [ 42 m</td>
<td>CSI 42 m</td>
<td>Green background, compliant with ISO Standard 6429.</td>
</tr>
<tr>
<td>ESC [ 43 m</td>
<td>CSI 43 m</td>
<td>Yellow background, compliant with ISO Standard 6429.</td>
</tr>
<tr>
<td>ESC [ 44 m</td>
<td>CSI 44 m</td>
<td>Blue background, compliant with ISO Standard 6429.</td>
</tr>
<tr>
<td>ESC [ 45 m</td>
<td>CSI 45 m</td>
<td>Magenta background, compliant with ISO Standard 6429.</td>
</tr>
<tr>
<td>ESC [ 46 m</td>
<td>CSI 46 m</td>
<td>Cyan background, compliant with ISO Standard 6429.</td>
</tr>
<tr>
<td>ESC [ 47 m</td>
<td>CSI 47 m</td>
<td>White background, compliant with ISO Standard 6429.</td>
</tr>
<tr>
<td>ESC [ = 3 h</td>
<td>CSI = 3 h</td>
<td>Set Mode 80x25 color.</td>
</tr>
<tr>
<td>ESC [ row;col H</td>
<td>CSI row;col H</td>
<td>Set cursor position to row;col. Row and col are strings of ASCII digits.</td>
</tr>
</tbody>
</table>
### B.3 Example Keyboard Layout

<table>
<thead>
<tr>
<th>Usage</th>
<th>EFI_KEY enum value</th>
<th>USB Keyboard HID Values</th>
<th>Typical AT-101 key position</th>
</tr>
</thead>
<tbody>
<tr>
<td>Key a and A</td>
<td>EfiKeyC1</td>
<td>0x04 31</td>
<td></td>
</tr>
<tr>
<td>Key b and B</td>
<td>EfiKeyB5</td>
<td>0x05 50</td>
<td></td>
</tr>
<tr>
<td>Key c and C</td>
<td>EfiKeyB3</td>
<td>0x06 48</td>
<td></td>
</tr>
<tr>
<td>Key d and D</td>
<td>EfiKeyC3</td>
<td>0x07 33</td>
<td></td>
</tr>
<tr>
<td>Key e and E</td>
<td>EfiKeyD3</td>
<td>0x08 19</td>
<td></td>
</tr>
<tr>
<td>Key f and F</td>
<td>EfiKeyC4</td>
<td>0x09 34</td>
<td></td>
</tr>
<tr>
<td>Key g and G</td>
<td>EfiKeyC5</td>
<td>0x0A 35</td>
<td></td>
</tr>
<tr>
<td>Key h and H</td>
<td>EfiKeyC6</td>
<td>0x0B 36</td>
<td></td>
</tr>
<tr>
<td>Key i and I</td>
<td>EfiKeyD8</td>
<td>0x0C 24</td>
<td></td>
</tr>
<tr>
<td>Key j and J</td>
<td>EfiKeyC7</td>
<td>0x0D 37</td>
<td></td>
</tr>
<tr>
<td>Key k and K</td>
<td>EfiKeyC8</td>
<td>0x0E 38</td>
<td></td>
</tr>
<tr>
<td>Key l and L</td>
<td>EfiKeyC9</td>
<td>0x0F 39</td>
<td></td>
</tr>
<tr>
<td>Key m and M</td>
<td>EfiKeyB7</td>
<td>0x10 52</td>
<td></td>
</tr>
<tr>
<td>Key n and N</td>
<td>EfiKeyB6</td>
<td>0x11 51</td>
<td></td>
</tr>
<tr>
<td>Key o and O</td>
<td>EfiKeyD9</td>
<td>0x12 25</td>
<td></td>
</tr>
<tr>
<td>Key p and p</td>
<td>EfiKeyD10</td>
<td>0x13 26</td>
<td></td>
</tr>
<tr>
<td>Key q and Q</td>
<td>EfiKeyD1</td>
<td>0x14 17</td>
<td></td>
</tr>
<tr>
<td>Key r and R</td>
<td>EfiKeyD4</td>
<td>0x15 20</td>
<td></td>
</tr>
<tr>
<td>Key s and S</td>
<td>EfiKeyC2</td>
<td>0x16 32</td>
<td></td>
</tr>
<tr>
<td>Key t and T</td>
<td>EfiKeyD5</td>
<td>0x17 21</td>
<td></td>
</tr>
<tr>
<td>Key u and U</td>
<td>EfiKeyD7</td>
<td>0x18 23</td>
<td></td>
</tr>
<tr>
<td>Key v and V</td>
<td>EfiKeyB4</td>
<td>0x19 49</td>
<td></td>
</tr>
<tr>
<td>Key w and W</td>
<td>EfiKeyD2</td>
<td>0x1A 18</td>
<td></td>
</tr>
<tr>
<td>Key x and X</td>
<td>EfiKeyB2</td>
<td>0x1B 47</td>
<td></td>
</tr>
<tr>
<td>Key y and Y</td>
<td>EfiKeyD6</td>
<td>0x1C 22</td>
<td></td>
</tr>
<tr>
<td>Key z and Z</td>
<td>EfiKeyB1</td>
<td>0x1D 46</td>
<td></td>
</tr>
<tr>
<td>Key 1 and !</td>
<td>EfiKeyE1</td>
<td>0x1E</td>
<td>2</td>
</tr>
<tr>
<td>------------</td>
<td>---------</td>
<td>------</td>
<td>---</td>
</tr>
<tr>
<td>Key 2 and @</td>
<td>EfiKeyE2</td>
<td>0x1F</td>
<td>3</td>
</tr>
<tr>
<td>Key 3 and #</td>
<td>EfiKeyE3</td>
<td>0x20</td>
<td>4</td>
</tr>
<tr>
<td>Key 4 and $</td>
<td>EfiKeyE4</td>
<td>0x21</td>
<td>5</td>
</tr>
<tr>
<td>Key 5 and %</td>
<td>EfiKeyE5</td>
<td>0x22</td>
<td>6</td>
</tr>
<tr>
<td>Key 6 and ^</td>
<td>EfiKeyE6</td>
<td>0x23</td>
<td>7</td>
</tr>
<tr>
<td>Key 7 and &amp;</td>
<td>EfiKeyE7</td>
<td>0x24</td>
<td>8</td>
</tr>
<tr>
<td>Key 8 and *</td>
<td>EfiKeyE8</td>
<td>0x25</td>
<td>9</td>
</tr>
<tr>
<td>Key 9 and (</td>
<td>EfiKeyE9</td>
<td>0x26</td>
<td>10</td>
</tr>
<tr>
<td>Key 0 and )</td>
<td>EfiKeyE10</td>
<td>0x27</td>
<td>11</td>
</tr>
<tr>
<td>Enter*</td>
<td>EfiKeyEnter</td>
<td>0x28</td>
<td>43</td>
</tr>
<tr>
<td>Escape*</td>
<td>EfiKeyEsc</td>
<td>0x29</td>
<td>110</td>
</tr>
<tr>
<td>Del*/Backspace*</td>
<td>EfiKeyBackSpace</td>
<td>0x2A</td>
<td>15</td>
</tr>
<tr>
<td>Tab*</td>
<td>EfiKeyTab</td>
<td>0x2B</td>
<td>16</td>
</tr>
<tr>
<td>Spacebar</td>
<td>EfiKeySpaceBar</td>
<td>0x2C</td>
<td>61</td>
</tr>
<tr>
<td>Key - and _</td>
<td>EfiKeySpaceBar</td>
<td>0x2D</td>
<td>12</td>
</tr>
<tr>
<td>Key = and +</td>
<td>EfiKeySpaceBar</td>
<td>0x2E</td>
<td>13</td>
</tr>
<tr>
<td>Key [ and {</td>
<td>EfiKeySpaceBar</td>
<td>0x2F</td>
<td>27</td>
</tr>
<tr>
<td>Key ] and }</td>
<td>EfiKeyD12</td>
<td>0x30</td>
<td>28</td>
</tr>
<tr>
<td>Key \ and</td>
<td>EfiKeyD13</td>
<td>0x31</td>
<td>29</td>
</tr>
<tr>
<td>Key ; and :</td>
<td>EfiKeyC10</td>
<td>0x33</td>
<td>40</td>
</tr>
<tr>
<td>Key ` and &quot;</td>
<td>EfiKeyC11</td>
<td>0x34</td>
<td>41</td>
</tr>
<tr>
<td>Key ` and ~</td>
<td>EfiKeyE0</td>
<td>0x35</td>
<td>1</td>
</tr>
<tr>
<td>Key , and &lt;</td>
<td>EfiKeyB8</td>
<td>0x36</td>
<td>53</td>
</tr>
<tr>
<td>Key . and &gt;</td>
<td>EfiKeyB9</td>
<td>0x37</td>
<td>54</td>
</tr>
<tr>
<td>Key / and ?</td>
<td>EfiKeyB10</td>
<td>0x38</td>
<td>55</td>
</tr>
<tr>
<td>Capslock*</td>
<td>EfiKeyCapsLock</td>
<td>0x39</td>
<td>30</td>
</tr>
<tr>
<td>F1*</td>
<td>EfiKeyF1</td>
<td>0x3A</td>
<td>112</td>
</tr>
<tr>
<td>F2*</td>
<td>EfiKeyF2</td>
<td>0x3B</td>
<td>113</td>
</tr>
<tr>
<td>F3*</td>
<td>EfiKeyF3</td>
<td>0x3C</td>
<td>114</td>
</tr>
<tr>
<td>Key</td>
<td>EfiKey</td>
<td>Code</td>
<td>Key</td>
</tr>
<tr>
<td>-------</td>
<td>---------</td>
<td>-------</td>
<td>-------</td>
</tr>
<tr>
<td>F4*</td>
<td>EfiKeyF4</td>
<td>0x3D</td>
<td>115</td>
</tr>
<tr>
<td>F5*</td>
<td>EfiKeyF5</td>
<td>0x3E</td>
<td>116</td>
</tr>
<tr>
<td>F6*</td>
<td>EfiKeyF6</td>
<td>0x3F</td>
<td>117</td>
</tr>
<tr>
<td>F7*</td>
<td>EfiKeyF7</td>
<td>0x40</td>
<td>118</td>
</tr>
<tr>
<td>F8*</td>
<td>EfiKeyF8</td>
<td>0x41</td>
<td>119</td>
</tr>
<tr>
<td>F9*</td>
<td>EfiKeyF9</td>
<td>0x42</td>
<td>120</td>
</tr>
<tr>
<td>F10*</td>
<td>EfiKeyF10</td>
<td>0x43</td>
<td>121</td>
</tr>
<tr>
<td>F11*</td>
<td>EfiKeyF11</td>
<td>0x44</td>
<td>122</td>
</tr>
<tr>
<td>F12*</td>
<td>EfiKeyF12</td>
<td>0x45</td>
<td>123</td>
</tr>
<tr>
<td>PrintScreen*</td>
<td>EfiKeyPrint</td>
<td>0x46</td>
<td>124</td>
</tr>
<tr>
<td>ScrollLock*</td>
<td>EfiKeySLck</td>
<td>0x47</td>
<td>125</td>
</tr>
<tr>
<td>Pause*</td>
<td>EfiKeyPause</td>
<td>0x48</td>
<td>126</td>
</tr>
<tr>
<td>Insert*</td>
<td>EfiKeyIns</td>
<td>0x49</td>
<td>75</td>
</tr>
<tr>
<td>Home*</td>
<td>EfiKeyHome</td>
<td>0x4A</td>
<td>80</td>
</tr>
<tr>
<td>PageUp*</td>
<td>EfiKeyPgUp</td>
<td>0x4B</td>
<td>85</td>
</tr>
<tr>
<td>Delete*</td>
<td>EfiKeyDel</td>
<td>0x4C</td>
<td>76</td>
</tr>
<tr>
<td>End*</td>
<td>EfiKeyEnd</td>
<td>0x4D</td>
<td>81</td>
</tr>
<tr>
<td>PageDown*</td>
<td>EfiKeyPgDn</td>
<td>0x4E</td>
<td>86</td>
</tr>
<tr>
<td>RightArrow*</td>
<td>EfiKeyRightArrow</td>
<td>0x4F</td>
<td>89</td>
</tr>
<tr>
<td>LeftArrow*</td>
<td>EfiKeyLeftArrow</td>
<td>0x50</td>
<td>79</td>
</tr>
<tr>
<td>DownArrow*</td>
<td>EfiKeyDownArrow</td>
<td>0x51</td>
<td>84</td>
</tr>
<tr>
<td>UpArrow*</td>
<td>EfiKeyUpArrow</td>
<td>0x52</td>
<td>83</td>
</tr>
<tr>
<td>NumLock*</td>
<td>EfiKeyNLck</td>
<td>0x53</td>
<td>90</td>
</tr>
<tr>
<td>Keypad /</td>
<td>EfiKeySlash</td>
<td>0x54</td>
<td>95</td>
</tr>
<tr>
<td>Keypad *</td>
<td>EfiKeyAsterisk</td>
<td>0x55</td>
<td>100</td>
</tr>
<tr>
<td>Keypad -</td>
<td>EfiKeyMinus</td>
<td>0x56</td>
<td>105</td>
</tr>
<tr>
<td>Keypad +</td>
<td>EfiKeyPlus</td>
<td>0x57</td>
<td>106</td>
</tr>
<tr>
<td>Keypad Enter*</td>
<td>EfiKeyEnter</td>
<td>0x58</td>
<td>108</td>
</tr>
<tr>
<td>Keypad 1 and End*</td>
<td>EfiKeyOne</td>
<td>0x59</td>
<td>93</td>
</tr>
<tr>
<td>Keypad 2 and DownArrow*</td>
<td>EfiKeyTwo</td>
<td>0x5A</td>
<td>98</td>
</tr>
</tbody>
</table>
An * indicates a non-printable character or keyboard behavior
Appendix C - Device Path Examples

This appendix presents an example EFI Device Path and explains its relationship to the ACPI name space. An example system design is presented along with its corresponding ACPI name space. These physical examples are mapped back to EFI Device Paths.

C.1 Example Computer System

Figure C-1 represents a hypothetical computer system architecture that will be used to discuss the construction of EFI Device Paths. The system consists of a memory controller that connects directly to the processors’ front side bus. The memory controller is only part of a larger chipset, and it connects to a root PCI host bridge chip, and a secondary root PCI host bridge chip. The secondary PCI host bridge chip produces a PCI bus that contains a PCI to PCI bridge. The root PCI host bridge produces a PCI bus, and also contains USB, ATA66, and AC ‘97 controllers. The root PCI host bridge also contains an LPC bus that is used to connect a SIO (Super IO) device. The SIO contains a PC-AT-compatible floppy disk controller, and other PC-AT-compatible devices like a keyboard controller.

Figure C-1 Example Computer System

The remainder of this appendix describes how to construct a device path for three example devices from the system in Figure C-1. The following is a list of the examples used:

- Legacy floppy
- IDE Disk
- Secondary root PCI bus with PCI to PCI bridge
**Figure C-2** is a partial ACPI name space for the system in Figure C-1. **Figure C-2** is based on Figure 5-3 in the *Advanced Configuration and Power Interface Specification*.

---

**C.2 Legacy Floppy**

The legacy floppy controller is contained in the SIO chip that is connected root PCI bus host bridge chip. The root PCI host bridge chip produces PCI bus 0, and other resources that appear directly to the processors in the system.

In ACPI this configuration is represented in the _SB, system bus tree, of the ACPI name space. PCI0 is a child of _SB and it represents the root PCI host bridge. The SIO appears to the system to be a set of ISA devices, so it is represented as a child of PCI0 with the name ISA0. The floppy controller is represented by FLPY as a child of the ISA0 bus.

The EFI Device Path for the legacy floppy is defined in **Table C-1**. It would contain entries for the following things:

- Root PCI Bridge. ACPI Device Path _HID PNP0A03, _UID 0. ACPI name space \_SB\PCI0
- PCI to ISA Bridge. PCI Device Path with device and function of the PCI to ISA bridge. ACPI name space \_SB\PCI0\ISA0
- Floppy Plug and Play ID. ACPI Device Path _HID PNP0303, _UID 0. ACPI name space \_SB\PCI0\ISA0\FLPY
- End Device Path
Table C-1 Legacy Floppy Device Path

<table>
<thead>
<tr>
<th>Byte Offset</th>
<th>Byte Length</th>
<th>Data</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>1</td>
<td>0x02</td>
<td><strong>Generic Device Path Header</strong> – Type ACPI Device Path</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>0x01</td>
<td>Sub type – ACPI Device Path</td>
</tr>
<tr>
<td>2</td>
<td>2</td>
<td>0x0C</td>
<td>Length</td>
</tr>
<tr>
<td>4</td>
<td>4</td>
<td>0x41D0, 0x0A03</td>
<td><strong>_HID PNP0A03</strong> – 0x41D0 represents the compressed string ‘PNP’ and is encoded in the low order bytes. The compression method is described in the ACPI Specification.</td>
</tr>
<tr>
<td>8</td>
<td>4</td>
<td>0x0000</td>
<td><strong>_UID</strong></td>
</tr>
<tr>
<td>C</td>
<td>1</td>
<td>0x01</td>
<td><strong>Generic Device Path Header</strong> – Type Hardware Device Path</td>
</tr>
<tr>
<td>D</td>
<td>1</td>
<td>0x01</td>
<td>Sub type – PCI Device Path</td>
</tr>
<tr>
<td>E</td>
<td>2</td>
<td>0x06</td>
<td>Length</td>
</tr>
<tr>
<td>10</td>
<td>1</td>
<td>0x00</td>
<td>PCI Function</td>
</tr>
<tr>
<td>11</td>
<td>1</td>
<td>0x10</td>
<td>PCI Device</td>
</tr>
<tr>
<td>12</td>
<td>1</td>
<td>0x02</td>
<td><strong>Generic Device Path Header</strong> – Type ACPI Device Path</td>
</tr>
<tr>
<td>13</td>
<td>1</td>
<td>0x01</td>
<td>Sub type – ACPI Device Path</td>
</tr>
<tr>
<td>14</td>
<td>2</td>
<td>0x0C</td>
<td>Length</td>
</tr>
<tr>
<td>16</td>
<td>4</td>
<td>0x41D0, 0x0303</td>
<td><strong>_HID PNP0303</strong></td>
</tr>
<tr>
<td>1A</td>
<td>4</td>
<td>0x0000</td>
<td><strong>_UID</strong></td>
</tr>
<tr>
<td>1E</td>
<td>1</td>
<td>0xFF</td>
<td><strong>Generic Device Path Header</strong> – Type End Device Path</td>
</tr>
<tr>
<td>1F</td>
<td>1</td>
<td>0xFF</td>
<td>Sub type – End Device Path</td>
</tr>
<tr>
<td>20</td>
<td>2</td>
<td>0x04</td>
<td>Length</td>
</tr>
</tbody>
</table>

C.3 IDE Disk

The IDE Disk controller is a PCI device that is contained in a function of the root PCI host bridge. The root PCI host bridge is a multi function device and has a separate function for chipset registers, USB, and IDE. The disk connected to the IDE ATA bus is defined as being on the primary or secondary ATA bus, and of being the master or slave device on that bus.

In ACPI this configuration is represented in the _SB, system bus tree, of the ACPI name space. PCI0 is a child of _SB and it represents the root PCI host bridge. The IDE controller appears to the system to be a PCI device with some legacy properties, so it is represented as a child of PCI0 with the name IDE0. PRIM is a child of IDE0 and it represents the primary ATA bus of the IDE controller. MAST is a child of PRIM and it represents that this device is the ATA master device on this primary ATA bus.

The EFI Device Path for the PCI IDE controller is defined in Table C-2. It would contain entries for the following things:

- Root PCI Bridge. ACPI Device Path _HID PNP0A03, _UID 0. ACPI name space \_SB\PCI0
- PCI IDE controller. PCI Device Path with device and function of the IDE controller. ACPI name space \_SB\PCI0\IDE0
• ATA Address. ATA Messaging Device Path for Primary bus and Master device. ACPI name space _SB\PCI0\IDE0\PRIM\MAST

• End Device Path

Table C-2 IDE Disk Device Path

<table>
<thead>
<tr>
<th>Byte Offset</th>
<th>Byte Length</th>
<th>Data</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>1</td>
<td>0x02</td>
<td>Generic Device Path Header – Type ACPI Device Path</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>0x01</td>
<td>Sub type – ACPI Device Path</td>
</tr>
<tr>
<td>2</td>
<td>2</td>
<td>0x0C</td>
<td>Length</td>
</tr>
<tr>
<td>4</td>
<td>4</td>
<td>0x41D0, 0x0A03</td>
<td>_HID PNP0A03 – 0x41D0 represents the compressed string ‘PNP’ and is encoded in the low order bytes. The compression method is described in the ACPI Specification.</td>
</tr>
<tr>
<td>8</td>
<td>4</td>
<td>0x0000</td>
<td>_UID</td>
</tr>
<tr>
<td>C</td>
<td>1</td>
<td>0x01</td>
<td>Generic Device Path Header – Type Hardware Device Path</td>
</tr>
<tr>
<td>D</td>
<td>1</td>
<td>0x01</td>
<td>Sub type PCI Device Path</td>
</tr>
<tr>
<td>E</td>
<td>2</td>
<td>0x06</td>
<td>Length</td>
</tr>
<tr>
<td>10</td>
<td>1</td>
<td>0x01</td>
<td>PCI Function</td>
</tr>
<tr>
<td>11</td>
<td>1</td>
<td>0x10</td>
<td>PCI Device</td>
</tr>
<tr>
<td>12</td>
<td>1</td>
<td>0x03</td>
<td>Generic Device Path Header – Messaging Device Path</td>
</tr>
<tr>
<td>13</td>
<td>1</td>
<td>0x01</td>
<td>Sub type – ATAPI Device Path</td>
</tr>
<tr>
<td>14</td>
<td>2</td>
<td>0x06</td>
<td>Length</td>
</tr>
<tr>
<td>16</td>
<td>1</td>
<td>0x00</td>
<td>Primary =0, Secondary = 1</td>
</tr>
<tr>
<td>17</td>
<td>1</td>
<td>0x00</td>
<td>Master = 0, Slave = 1</td>
</tr>
<tr>
<td>18</td>
<td>2</td>
<td>0x0000</td>
<td>LUN</td>
</tr>
<tr>
<td>1A</td>
<td>1</td>
<td>0xFF</td>
<td>Generic Device Path Header – Type End Device Path</td>
</tr>
<tr>
<td>1B</td>
<td>1</td>
<td>0xFF</td>
<td>Sub type – End Device Path</td>
</tr>
<tr>
<td>1C</td>
<td>2</td>
<td>0x04</td>
<td>Length</td>
</tr>
</tbody>
</table>

C.4 Secondary Root PCI Bus with PCI to PCI Bridge

The secondary PCI host bridge materializes a second set of PCI buses into the system. The PCI buses on the secondary PCI host bridge are totally independent of the PCI buses on the root PCI host bridge. The only relationship between the two is they must be configured to not consume the same resources. The primary PCI bus of the secondary PCI host bridge also contains a PCI to PCI bridge. There is some arbitrary PCI device plugged in behind the PCI to PCI bridge in a PCI slot.

In ACPI this configuration is represented in the _SB, system bus tree, of the ACPI name space. PCI1 is a child of _SB and it represents the secondary PCI host bridge. The PCI to PCI bridge and the device plugged into the slot on its primary bus are not described in the ACPI name space. These devices can be fully configured by following the applicable PCI specification.

The EFI Device Path for the secondary root PCI bridge with a PCI to PCI bridge is defined in Table C-3. It would contain entries for the following things:
• Root PCI Bridge. ACPI Device Path _HID PNP0A03, _UID 1. ACPI name space \SB\PCI1
• PCI to PCI Bridge. PCI Device Path with device and function of the PCI Bridge. ACPI name space \SB\PCI1, PCI to PCI bridges are defined by PCI specification and not ACPI.
• PCI Device. PCI Device Path with the device and function of the PCI device. ACPI name space \SB\PCI1, PCI devices are defined by PCI specification and not ACPI.
• End Device Path.

Table C-3 Secondary Root PCI Bus with PCI to PCI Bridge Device Path

<table>
<thead>
<tr>
<th>Byte Offset</th>
<th>Byte Length</th>
<th>Data</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>1</td>
<td>0x02</td>
<td><em>Generic Device Path Header</em> – Type ACPI Device Path</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>0x01</td>
<td>Sub type – ACPI Device Path</td>
</tr>
<tr>
<td>2</td>
<td>2</td>
<td>0x0C</td>
<td>Length</td>
</tr>
<tr>
<td>4</td>
<td>4</td>
<td>0x41D0, 0x0A03</td>
<td>_HID PNP0A03 – 0x41D0 represents the compressed string ‘PNP’ and is encoded in the low order bytes. The compression method is described in the ACPI Specification.</td>
</tr>
<tr>
<td>8</td>
<td>4</td>
<td>0x0001</td>
<td>_UID</td>
</tr>
<tr>
<td>C</td>
<td>1</td>
<td>0x01</td>
<td><em>Generic Device Path Header</em> – Type Hardware Device Path</td>
</tr>
<tr>
<td>D</td>
<td>1</td>
<td>0x01</td>
<td>Sub type PCI Device Path</td>
</tr>
<tr>
<td>E</td>
<td>2</td>
<td>0x06</td>
<td>Length</td>
</tr>
<tr>
<td>10</td>
<td>1</td>
<td>0x00</td>
<td>PCI Function for PCI to PCI bridge</td>
</tr>
<tr>
<td>11</td>
<td>1</td>
<td>0x0c</td>
<td>PCI Device for PCI to PCI bridge</td>
</tr>
<tr>
<td>12</td>
<td>1</td>
<td>0x01</td>
<td><em>Generic Device Path Header</em> – Type Hardware Device Path</td>
</tr>
<tr>
<td>13</td>
<td>1</td>
<td>0x01</td>
<td>Sub type PCI Device Path</td>
</tr>
<tr>
<td>14</td>
<td>2</td>
<td>0x08</td>
<td>Length</td>
</tr>
<tr>
<td>16</td>
<td>1</td>
<td>0x00</td>
<td>PCI Function for PCI Device</td>
</tr>
<tr>
<td>17</td>
<td>1</td>
<td>0x00</td>
<td>PCI Device for PCI Device</td>
</tr>
<tr>
<td>18</td>
<td>1</td>
<td>0xFF</td>
<td><em>Generic Device Path Header</em> – Type End Device Path</td>
</tr>
<tr>
<td>19</td>
<td>1</td>
<td>0xFF</td>
<td>Sub type – End Device Path</td>
</tr>
<tr>
<td>1A</td>
<td>2</td>
<td>0x04</td>
<td>Length</td>
</tr>
</tbody>
</table>
C.5 ACPI Terms

Names in the ACPI name space that start with an underscore (“_”) are reserved by the ACPI specification and have architectural meaning. All ACPI names in the name space are four characters in length. The following four ACPI names are used in this specification.

_ADR. The Address on a bus that has standard enumeration. An example would be PCI, where the enumeration method is described in the PCI Local Bus specification.

_CRS. The current resource setting of a device. A _CRS is required for devices that are not enumerated in a standard fashion. _CRS is how ACPI converts nonstandard devices into Plug and Play devices.

_HID. Represents a device’s Plug and Play hardware ID, stored as a 32-bit compressed EISA ID. _HID objects are optional in ACPI. However, a _HID object must be used to describe any device that will be enumerated by the ACPI driver in the OS. This is how ACPI deals with non–Plug and Play devices.

_UID. Is a serial number style ID that does not change across reboots. If a system contains more than one device that reports the same _HID, each device must have a unique _UID. The _UID only needs to be unique for device that have the exact same _HID value.

C.6 EFI Device Path as a Name Space

Figure C-3 shows the EFI Device Path for the example system represented as a name space. The Device Path can be represented as a name space, but EFI does support manipulating the Device Path as a name space. You can only access Device Path information by locating the DEVICE_PATH_INTERFACE from a handle. Not all the nodes in a Device Path will have a handle.
Appendix D - Status Codes

EFI interfaces return an **EFI_STATUS** code. Table D-2, Table D-3, and Table D-4 list these codes for success, errors, and warnings, respectively. The range of status codes that have the highest bit set and the next to highest bit clear are reserved for use by EFI. The range of status codes that have both the highest bit set and the next to highest bit set are reserved for use by OEMs. Success and warning codes have their highest bit clear, so all success and warning codes have positive values. The range of status codes that have both the highest bit clear and the next to highest bit clear are reserved for use by EFI. The range of status code that have the highest bit clear and the next to highest bit set are reserved for use by OEMs. **Table D-1** lists the status code ranges described above.

### Table D-1 EFI_STATUS Code Ranges

<table>
<thead>
<tr>
<th>Supported 32-bit Range</th>
<th>Supported 64-bit Architecture Ranges</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x00000000-0x1fffffff</td>
<td>0x0000000000000000-0x1fffffffffffff</td>
<td>Warning codes reserved for use by UEFI main specification.</td>
</tr>
<tr>
<td>0x20000000-0x3ff</td>
<td>0x2000000000000000-0x3fffffff</td>
<td>Warning codes reserved for use by the Platform Initialization Architecture Specification.</td>
</tr>
<tr>
<td>0x40000000-0x7ff</td>
<td>0x4000000000000000-0x7fffffffffffff</td>
<td>Warning codes reserved for OEM usage.</td>
</tr>
<tr>
<td>0x80000000-0x9ff</td>
<td>0x8000000000000000-0x9fffffffffffff</td>
<td>Error codes reserved for use by UEFI main spec.</td>
</tr>
<tr>
<td>0xa0000000-0xbff</td>
<td>0xa000000000000000-0xbfffffffffffff</td>
<td>Error codes reserved for use by the Platform Initialization Architecture Specification.</td>
</tr>
<tr>
<td>0xc0000000-0xff</td>
<td>0xc000000000000000-0xcfffffffffffff</td>
<td>Error codes reserved for OEM usage.</td>
</tr>
</tbody>
</table>

### Table D-2 EFI_STATUS Success Codes (High Bit Clear)

<table>
<thead>
<tr>
<th>Mnemonic</th>
<th>Value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>EFI_SUCCESS</td>
<td>0</td>
<td>The operation completed successfully.</td>
</tr>
</tbody>
</table>

### Table D-3 EFI_STATUS Error Codes (High Bit Set)

<table>
<thead>
<tr>
<th>Mnemonic</th>
<th>Value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>EFI_LOAD_ERROR</td>
<td>1</td>
<td>The image failed to load.</td>
</tr>
<tr>
<td>EFI_INVALID_PARAMETER</td>
<td>2</td>
<td>A parameter was incorrect.</td>
</tr>
<tr>
<td>EFI_UNSUPPORTED</td>
<td>3</td>
<td>The operation is not supported.</td>
</tr>
<tr>
<td>EFI_BAD_BUFFER_SIZE</td>
<td>4</td>
<td>The buffer was not the proper size for the request.</td>
</tr>
</tbody>
</table>
### Mnemonic Value Description

<table>
<thead>
<tr>
<th>Mnemonic</th>
<th>Value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>EFI_BUFFER_TOO_SMALL</td>
<td>5</td>
<td>The buffer is not large enough to hold the requested data. The required buffer size is returned in the appropriate parameter when this error occurs.</td>
</tr>
<tr>
<td>EFI_NOT_READY</td>
<td>6</td>
<td>There is no data pending upon return.</td>
</tr>
<tr>
<td>EFI_DEVICE_ERROR</td>
<td>7</td>
<td>The physical device reported an error while attempting the operation.</td>
</tr>
<tr>
<td>EFI_WRITE_PROTECTED</td>
<td>8</td>
<td>The device cannot be written to.</td>
</tr>
<tr>
<td>EFI_OUT_OF_RESOURCES</td>
<td>9</td>
<td>A resource has run out.</td>
</tr>
<tr>
<td>EFI_VOLUME_CORRUPTED</td>
<td>10</td>
<td>An inconstancy was detected on the file system causing the operating to fail.</td>
</tr>
<tr>
<td>EFI_VOLUME_FULL</td>
<td>11</td>
<td>There is no more space on the file system.</td>
</tr>
<tr>
<td>EFI_NO_MEDIA</td>
<td>12</td>
<td>The device does not contain any medium to perform the operation.</td>
</tr>
<tr>
<td>EFI_MEDIA_CHANGED</td>
<td>13</td>
<td>The medium in the device has changed since the last access.</td>
</tr>
<tr>
<td>EFI_NOT_FOUND</td>
<td>14</td>
<td>The item was not found.</td>
</tr>
<tr>
<td>EFI_ACCESS_DENIED</td>
<td>15</td>
<td>Access was denied.</td>
</tr>
<tr>
<td>EFI_NO_RESPONSE</td>
<td>16</td>
<td>The server was not found or did not respond to the request.</td>
</tr>
<tr>
<td>EFI_NO_MAPPING</td>
<td>17</td>
<td>A mapping to a device does not exist.</td>
</tr>
<tr>
<td>EFI_TIMEOUT</td>
<td>18</td>
<td>The timeout time expired.</td>
</tr>
<tr>
<td>EFI_NOT_STARTED</td>
<td>19</td>
<td>The protocol has not been started.</td>
</tr>
<tr>
<td>EFI_ALREADY_STARTED</td>
<td>20</td>
<td>The protocol has already been started.</td>
</tr>
<tr>
<td>EFI_ABORTED</td>
<td>21</td>
<td>The operation was aborted.</td>
</tr>
<tr>
<td>EFI_ICMP_ERROR</td>
<td>22</td>
<td>An ICMP error occurred during the network operation.</td>
</tr>
<tr>
<td>EFI_TFTP_ERROR</td>
<td>23</td>
<td>A TFTP error occurred during the network operation.</td>
</tr>
<tr>
<td>EFI_PROTOCOL_ERROR</td>
<td>24</td>
<td>A protocol error occurred during the network operation.</td>
</tr>
<tr>
<td>EFI_INCOMPATIBLE_VERSION</td>
<td>25</td>
<td>The function encountered an internal version that was incompatible with a version requested by the caller.</td>
</tr>
<tr>
<td>EFI_SECURITY_VIOLATION</td>
<td>26</td>
<td>The function was not performed due to a security violation.</td>
</tr>
<tr>
<td>EFI_CRC_ERROR</td>
<td>27</td>
<td>A CRC error was detected.</td>
</tr>
<tr>
<td>EFI_END_OF_MEDIA</td>
<td>28</td>
<td>Beginning or end of media was reached</td>
</tr>
<tr>
<td>EFI_END_OF_FILE</td>
<td>31</td>
<td>The end of the file was reached.</td>
</tr>
<tr>
<td>EFI_INVALID_LANGUAGE</td>
<td>32</td>
<td>The language specified was invalid.</td>
</tr>
<tr>
<td>EFI_COMPROMISED_DATA</td>
<td>33</td>
<td>The security status of the data is unknown or compromised and the data must be updated or replaced to restore a valid security status.</td>
</tr>
<tr>
<td>EFI_IP_ADDRESS_CONFLICT</td>
<td>34</td>
<td>There is an address conflict address allocation</td>
</tr>
<tr>
<td>EFI_HTTP_ERROR</td>
<td>35</td>
<td>A HTTP error occurred during the network operation.</td>
</tr>
</tbody>
</table>
Table D-4 EFI_STATUS Warning Codes (High Bit Clear)

<table>
<thead>
<tr>
<th>Mnemonic</th>
<th>Value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>EFI_WARN_UNKNOWN_GLYPH</td>
<td>1</td>
<td>The string contained one or more characters that the device could not render and were skipped.</td>
</tr>
<tr>
<td>EFI_WARN_DELETE_FAILURE</td>
<td>2</td>
<td>The handle was closed, but the file was not deleted.</td>
</tr>
<tr>
<td>EFI_WARN_WRITE_FAILURE</td>
<td>3</td>
<td>The handle was closed, but the data to the file was not flushed properly.</td>
</tr>
<tr>
<td>EFI_WARN_BUFFER_TOO_SMALL</td>
<td>4</td>
<td>The resulting buffer was too small, and the data was truncated to the buffer size.</td>
</tr>
<tr>
<td>EFI_WARN_STALE_DATA</td>
<td>5</td>
<td>The data has not been updated within the timeframe set by local policy for this type of data.</td>
</tr>
<tr>
<td>EFI_WARN_FILE_SYSTEM</td>
<td>6</td>
<td>The resulting buffer contains UEFI-compliant file system.</td>
</tr>
<tr>
<td>EFI_WARN_RESET_REQUIRED</td>
<td>7</td>
<td>The operation will be processed across a system reset.</td>
</tr>
</tbody>
</table>
Appendix E - Universal Network Driver Interfaces

E.1 Introduction

This appendix defines the 32/64-bit H/W and S/W Universal Network Driver Interfaces (UNDIs). These interfaces provide one method for writing a network driver; other implementations are possible.

E.1.1 Definitions

Table E-1 Definitions

<table>
<thead>
<tr>
<th>Term</th>
<th>Definition</th>
</tr>
</thead>
<tbody>
<tr>
<td>BC</td>
<td>BaseCode</td>
</tr>
<tr>
<td></td>
<td>The PXE BaseCode, included as a core protocol in EFI, is comprised of a simple network stack (UDP/IP) and a few common network protocols (DHCP, Bootserver Discovery, TFTP) that are useful for remote booting machines.</td>
</tr>
<tr>
<td>LOM</td>
<td>LAN On Motherboard</td>
</tr>
<tr>
<td></td>
<td>This is a network device that is built onto the motherboard (or baseboard) of the machine.</td>
</tr>
<tr>
<td>NBP</td>
<td>Network Bootstrap Program</td>
</tr>
<tr>
<td></td>
<td>This is the first program that is downloaded into a machine that has selected a PXE capable device for remote boot services.</td>
</tr>
<tr>
<td></td>
<td>A typical NBP examines the machine it is running on to try to determine if the machine is capable of running the next layer (OS or application). If the machine is not capable of running the next layer, control is returned to the EFI boot manager and the next boot device is selected. If the machine is capable, the next layer is downloaded and control can then be passed to the downloaded program. Though most NBPs are OS loaders, NBPs can be written to be standalone applications such as diagnostics, backup/restore, remote management agents, browsers, etc.</td>
</tr>
<tr>
<td>NIC</td>
<td>Network Interface Card</td>
</tr>
<tr>
<td></td>
<td>Technically, this is a network device that is inserted into a bus on the motherboard or in an expansion board. For the purposes of this document, the term NIC will be used in a generic sense, meaning any device that enables a network connection (including LOMs and network devices on external busses (USB, 1394, etc.)).</td>
</tr>
<tr>
<td>ROM</td>
<td>Read-Only Memory</td>
</tr>
<tr>
<td></td>
<td>When used in this specification, ROM refers to a nonvolatile memory storage device on a NIC.</td>
</tr>
</tbody>
</table>
When implementing PXE services, protocols, ROMs or drivers, it is a good idea to understand the related network protocols and BIOS specifications. **Table E-2** below includes all of the specifications referenced in this document.

**Table E-2 Referenced Specifications**

<table>
<thead>
<tr>
<th>Acronym</th>
<th>Protocol/Specification</th>
</tr>
</thead>
</table>
| Assigned Numbers | Lists the reserved numbers used in the RFCs and in this specification. See “Links to UEFI-Related Documents” ([http://uefi.org/uefi](http://uefi.org/uefi)) under the heading “Assigned Numbers”.
<p>| BIOS          | Basic Input/Output System – Contact your BIOS manufacturer for reference and programming manuals. |</p>
<table>
<thead>
<tr>
<th>Acronym</th>
<th>Protocol/Specification</th>
</tr>
</thead>
</table>
| BOOTP   | **Bootstrap Protocol** –  
  - See “Links to UEFI-Related Documents” ([http://uefi.org/uefi](http://uefi.org/uefi)) under the heading “Bootstrap Protocol (BOOTP)”.  
  These references are included for backward compatibility. BC protocol supports DHCP and BOOTP:  
  - See “Links to UEFI-Related Documents” ([http://uefi.org/uefi](http://uefi.org/uefi)) under the heading “BOOTP Clarifications and Extensions”.  
  - See “Links to UEFI-Related Documents” ([http://uefi.org/uefi](http://uefi.org/uefi)) under the heading “Bootstrap Protocol (BOOTP) Interoperation Between DHCP and BOOTP”.
  Required reading for those implementing the PXE Base Code Protocol BC protocol or PXE Boot servers. |
| DHCP    | **Dynamic Host Configuration Protocol**  
  - See “Links to UEFI-Related Documents” ([http://uefi.org/uefi](http://uefi.org/uefi)) under the heading “DHCP”.  
  - See “Links to UEFI-Related Documents” ([http://uefi.org/uefi](http://uefi.org/uefi)) under the heading “Index of RFC (IETF)”.  
  - See “Links to UEFI-Related Documents” ([http://uefi.org/uefi](http://uefi.org/uefi)) under the heading “DHCP Reconfigure Extension”.  
  - See “Links to UEFI-Related Documents” ([http://uefi.org/uefi](http://uefi.org/uefi)) under the heading “DHCP for Ipv4”.  
  - See “Links to UEFI-Related Documents” ([http://uefi.org/uefi](http://uefi.org/uefi)) under the heading “Interoperations between DHCP and BOOTP”.
  Required reading for those implementing the PXE Base Code Protocol or PXE Boot servers. |
| EFI     | **Extensible Firmware Interface**  
  - See “Links to UEFI-Related Documents” ([http://uefi.org/uefi](http://uefi.org/uefi)) under the heading “Intel Developer Centers”.  
  Required reading for those implementing NBPs, OS loaders and preboot applications for machines with the EFI preboot environment. |
| ICMP    | **Internet Control Message Protocol**  
  - See “Links to UEFI-Related Documents” ([http://uefi.org/uefi](http://uefi.org/uefi)) under the heading “ICMP for Ipv4”.  
  - See “Links to UEFI-Related Documents” ([http://uefi.org/uefi](http://uefi.org/uefi)) under the heading “ICMP for Ipv6”.  
  Required reading for those implementing the BC protocol. |
| IETF    | Internet Engineering Task Force  
  - See “Links to UEFI-Related Documents” ([http://uefi.org/uefi](http://uefi.org/uefi)) under the heading “Internet Engineering Task Force (IETF)”.  
  This is a good starting point for obtaining electronic copies of Internet standards, drafts, and RFCs. |
| IGMP    | **Internet Group Management Protocol**  
  - See “Links to UEFI-Related Documents” ([http://uefi.org/uefi](http://uefi.org/uefi)) under the heading “Internet Group Management Protocol”.
  Required reading for those implementing the PXE Base Code Protocol. |
| IP      | Internet Protocol  
  - See “Links to UEFI-Related Documents” ([http://uefi.org/uefi](http://uefi.org/uefi)) under the heading “Ipv4”.  
  - See “Links to UEFI-Related Documents” ([http://uefi.org/uefi](http://uefi.org/uefi)) under the heading “Ipv6”.  
  Required reading for those implementing the BC protocol. |
| MTFTP   | **Multicast TFTP** – Defined in the 16-bit PXE specification.  
  Required reading for those implementing the PXE Base Code Protocol. |
<table>
<thead>
<tr>
<th>Acronym</th>
<th>Protocol/Specification</th>
</tr>
</thead>
<tbody>
<tr>
<td>PCI</td>
<td>Peripheral Component Interface</td>
</tr>
<tr>
<td></td>
<td>• See “Links to UEFI-Related Documents” (<a href="http://uefi.org/uefi">http://uefi.org/uefi</a>) under the heading “Peripheral Component Interface (PCI)”. Source for PCI specifications. Required reading for those implementing S/W or H/W UNDI on a PCI NIC or LOM.</td>
</tr>
<tr>
<td></td>
<td>• See “Links to UEFI-Related Documents” (<a href="http://uefi.org/uefi">http://uefi.org/uefi</a>) under the heading “Plug and Play”. Source for PnP specifications.</td>
</tr>
<tr>
<td>PXE</td>
<td>Preboot eXecution Environment</td>
</tr>
<tr>
<td></td>
<td>16-bit PXE v2.1:</td>
</tr>
<tr>
<td></td>
<td>• See “Links to UEFI-Related Documents” (<a href="http://uefi.org/uefi">http://uefi.org/uefi</a>) under the heading “Preboot eXecution Environment (PXE)”. Required reading.</td>
</tr>
<tr>
<td>RFC</td>
<td>Request For Comments –</td>
</tr>
<tr>
<td></td>
<td>• See “Links to UEFI-Related Documents” (<a href="http://uefi.org/uefi">http://uefi.org/uefi</a>) under the heading “Request for Comments”.</td>
</tr>
<tr>
<td>TCP</td>
<td>Transmission Control Protocol</td>
</tr>
<tr>
<td></td>
<td>• See “Links to UEFI-Related Documents” (<a href="http://uefi.org/uefi">http://uefi.org/uefi</a>) under the heading “TCPv4”.</td>
</tr>
<tr>
<td></td>
<td>• See “Links to UEFI-Related Documents” (<a href="http://uefi.org/uefi">http://uefi.org/uefi</a>) under the heading “TCPv6”. Required reading for those implementing the PXE Base Code Protocol.</td>
</tr>
<tr>
<td>TFTP</td>
<td>Trivial File Transfer Protocol</td>
</tr>
<tr>
<td></td>
<td>TFTP</td>
</tr>
<tr>
<td></td>
<td>• See “Links to UEFI-Related Documents” (<a href="http://uefi.org/uefi">http://uefi.org/uefi</a>) under the heading “TFTP Protocol”.</td>
</tr>
<tr>
<td></td>
<td>• See “Links to UEFI-Related Documents” (<a href="http://uefi.org/uefi">http://uefi.org/uefi</a>) under the heading “TFTP Option Extension”.</td>
</tr>
<tr>
<td></td>
<td>• See “Links to UEFI-Related Documents” (<a href="http://uefi.org/uefi">http://uefi.org/uefi</a>) under the heading “TFTP Blocksize Option”.</td>
</tr>
<tr>
<td></td>
<td>• See “Links to UEFI-Related Documents” (<a href="http://uefi.org/uefi">http://uefi.org/uefi</a>) under the heading “TFTP Timeout Interval and Transfer Size Options”. Required reading for those implementing the PXE Base Code Protocol.</td>
</tr>
<tr>
<td>UDP</td>
<td>User Datagram Protocol</td>
</tr>
<tr>
<td></td>
<td>• See “Links to UEFI-Related Documents” (<a href="http://uefi.org/uefi">http://uefi.org/uefi</a>) under the heading “UDP over IPv4”.</td>
</tr>
<tr>
<td></td>
<td>• See “Links to UEFI-Related Documents” (<a href="http://uefi.org/uefi">http://uefi.org/uefi</a>) under the heading “UDP over IPv6”. Required reading for those implementing the PXE Base Code Protocol.</td>
</tr>
<tr>
<td>WfM</td>
<td>Wired for Management</td>
</tr>
<tr>
<td></td>
<td>• See “Links to UEFI-Related Documents” (<a href="http://uefi.org/uefi">http://uefi.org/uefi</a>) under the heading “Wired for Management”. Recommended reading for those implementing the PXE Base Code Protocol or PXE Bootservers.</td>
</tr>
</tbody>
</table>
E.1.3 OS Network Stacks

This is a simplified overview of three OS network stacks that contain three types of network drivers: Custom, S/W UNDI and H/W UNDI. Figure E-1 depicts an application bound to an OS protocol stack, which is in turn bound to a protocol driver that is bound to three NICs. Table E-3 below gives a brief list of pros and cons about each type of driver implementation.

![Figure E-1 Network Stacks with Three Classes of Drivers](image)

### Table E-3 Driver Types: Pros and Cons

<table>
<thead>
<tr>
<th>Driver</th>
<th>Pro</th>
<th>Con</th>
</tr>
</thead>
</table>
| Custom   | • Can be very fast and efficient. NIC vendor tunes driver to OS & device.  
|          | • OS vendor does not have to write NIC driver.                        | • New driver for each OS/architecture must be maintained by NIC vendor.  
|          |                                                                      | • OS vendor must trust code supplied by third-party.                 |
|          |                                                                      | • OS vendor cannot test all possible driver/NIC versions.            |
|          |                                                                      | • Driver must be installed before NIC can be used.                   |
|          |                                                                      | • Possible performance sink if driver is poorly written.             |
|          |                                                                      | • Possible security risk if driver has back door.                    |
| S/W UNDI | • S/W UNDI driver is simpler than a Custom driver. Easier to test outside of the OS environment.  
|          | • OS vendor can tune the universal protocol driver for best OS performance.  
|          | • NIC vendor only has to write one driver per processor architecture. | • Slightly slower than Custom or H/W UNDI because of extra call layer between protocol stack and NIC.  
|          |                                                                      | • S/W UNDI driver must be loaded before NIC can be used.             |
|          |                                                                      | • OS vendor has to write the universal driver.                       |
|          |                                                                      | • Possible performance sink if driver is poorly written.             |
|          |                                                                      | • Possible security risk if driver has back door.                    |
E.2 Overview

There are three major design changes between this specification and the 16-bit UNDI in version 2.1 of the PXE Specification:

- A new architectural hardware interface has been added.
- All UNDI commands use the same command format.
- BC is no longer part of the UNDI ROM.

E.2.1 32/64-bit UNDI Interface

The !PXE structures are used to locate and identify the type of 32/64-bit UNDI interface (H/W or S/W), as shown in Figure E-2. These structures are normally only used by the system BIOS and universal network drivers.

<table>
<thead>
<tr>
<th>Driver</th>
<th>Pro</th>
<th>Con</th>
</tr>
</thead>
</table>
| H/W UNDI | • H/W UNDI provides a common architectural interface to all network devices.  
• OS vendor controls all security and performance issues in network stack.  
• NIC vendor does not have to write any drivers.  
• NIC can be used without an OS or driver installed (preboot management). | • OS vendor has to write the universal driver (this might also be a Pro, depending on your point of view). |
The !PXE structures used for H/W and S/W UNDIs are similar but not identical. The difference in the
format is tied directly to the differences required by the implementation. The !PXE structures for 32/64-bit
UNDIs are not compatible with the !PXE structure for 16-bit UNDI.

The !PXE structure for H/W UNDI is built into the NIC hardware. The first nine fields (from offsets 0x00 to
0x0F) are implemented as read-only memory (or ports). The last three fields (from Len to Len + 0x0F) are
implemented as read/write memory (or ports). The optional reserved field at 0x10 is not defined in this
specification and may be used for vendor data.

The !PXE structure for S/W UNDI can be loaded into system memory from one of three places; ROM on a
NIC, system nonvolatile storage, or external storage. Since there are no direct memory or I/O ports
available in the S/W UNDI !PXE structure, an indirect callable entry point is provided. S/W UNDI
developers are free to make their internal designs as simple or complex as they desire, as long as all of
the UNDI commands in this specification are implemented.

Descriptions of the fields in the !PXE structures is given in Table E-4.

**Table E-4 !PXE Structure Field Definitions**

<table>
<thead>
<tr>
<th>Identifier</th>
<th>Value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Signature</td>
<td>“!PXE”</td>
<td>!PXE structure signature. This field is used to locate an UNDI hardware or software interface in system memory (or I/O) space. ‘!’ is in the first (lowest address) byte, ‘P’ is in the second byte, ‘X’ in the third and ‘E’ in the last. This field must be aligned on a 16-byte boundary (the last address byte must be zero).</td>
</tr>
<tr>
<td>Len</td>
<td>Varies</td>
<td>Number of !PXE structure bytes to checksum. When computing the checksum of this structure the Len field MUST be used as the number of bytes to checksum. The !PXE structure checksum is computed by adding all of the bytes in the structure, starting with the first byte of the structure Signature: ‘!’. If the 8-bit sum of all of the unsigned bytes in this structure is not zero, this is not a valid !PXE structure.</td>
</tr>
<tr>
<td>Fudge</td>
<td>Varies</td>
<td>This field is used to make the 8-bit checksum of this structure equal zero.</td>
</tr>
<tr>
<td>Rev</td>
<td>0x03</td>
<td>Revision of this structure.</td>
</tr>
<tr>
<td>IFcnt</td>
<td>Varies</td>
<td>This field reports the number (minus one) of physical external network connections that are controlled by this !PXE interface. (If there is one network connector, this field is zero. If there are two network connectors, this field is one.) For !PXE structure revision 0x03 or higher, in addition to this field, the value in IFcntExt field must be left-shifted by 8-bits and ORed with IFcnt to get the 16-bit value for the total number (minus one) of physical external network connections that are controlled by this !PXE interface.</td>
</tr>
</tbody>
</table>
| Major      | Varies| UNDI command interface. Minor revision number.  
**0x00 (Alpha):** This version of UNDI does not operate as a runtime driver. The callback interface defined in the UNDI Start command is required.  
**0x10 (Beta):** This version of UNDI can operate as an OS runtime driver. The callback interface defined in the UNDI Start command is required |
| Minor      | Varies| UNDI command interface. Minor revision number.  
**0x00 (Alpha):** This version of UNDI does not operate as a runtime driver. The callback interface defined in the UNDI Start command is required.  
**0x10 (Beta):** This version of UNDI can operate as an OS runtime driver. The callback interface defined in the UNDI Start command is required.
### IFcntExt

<table>
<thead>
<tr>
<th>Identifier</th>
<th>Value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>IFcntExt</td>
<td>Varies</td>
<td>If the IPXE structure revision 0x02 or earlier, this field is reserved and must be set to zero. If the IPXE structure revision 0x03 or higher, this field reports the upper 8-bits of the number of physical external network connections that is controlled by this IPXE interface.</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Identifier</th>
<th>Value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>reserved</td>
<td>0x00</td>
<td>This field is reserved and must be set to zero.</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Implementation</th>
<th>Varies</th>
<th>Identifies type of UNDI</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td>(S/W or H/W, 32 bit or 64 bit) and what features have been implemented. The implementation bits are defined below. Undefined bits must be set to zero by UNDI implementers. Applications/drivers must not rely on the contents of undefined bits (they may change later revisions).</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Bit 0x00: Command completion interrupts supported (1) or not supported (0)</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Bit 0x01: Packet received interrupts supported (1) or not supported (0)</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Bit 0x02: Transmit complete interrupts supported (1) or not supported (0)</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Bit 0x03: Software interrupt supported (1) or not supported (0)</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Bit 0x04: Filtered multicast receives supported (1) or not supported (0)</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Bit 0x05: Broadcast receives supported (1) or not supported (0)</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Bit 0x06: Promiscuous receives supported (1) or not supported (0)</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Bit 0x07: Promiscuous multicast receives supported (1) or not supported (0)</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Bit 0x08: Station MAC address settable (1) or not settable (0)</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Bit 0x09: Statistics supported (1) or not supported (0)</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Bit 0x0A,0x0B: NvData not available (0), read only (1), sparse write supported (2), bulk write supported (3)</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Bit 0x0C: Multiple frames per command supported (1) or not supported (0)</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Bit 0x0D: Command queuing supported (1) or not supported (0)</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Bit 0x0E: Command linking supported (1) or not supported (0)</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Bit 0x0F: Packet fragmenting supported (1) or not supported (0)</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Bit 0x10: Device can address 64 bits (1) or only 32 bits (0)</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Bit 0x1E: S/W UNDI: Entry point is virtual address (1) or unsigned offset from start of IPXE structure (0).</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Bit 0x1F: Interface type: H/W UNDI (1) or S/W UNDI (0)</td>
</tr>
</tbody>
</table>

### H/W UNDI Fields

<table>
<thead>
<tr>
<th>Identifier</th>
<th>Value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Reserved</td>
<td>Varies</td>
<td>This field is optional and may be used for OEM &amp; vendor unique data. If this field is present its length must be a multiple of 16 bytes and must be included in the IPXE structure checksum. This field, if present, will always start on a 16-byte boundary. <strong>Note:</strong> The size/contents of the IPXE structure may change in future revisions of this specification. Do not rely on OEM &amp; vendor data starting at the same offset from the beginning of the IPXE structure. It is recommended that the OEM &amp; vendor data include a signature that drivers/applications can search for.</td>
</tr>
</tbody>
</table>

Note: The size/contents of the IPXE structure may change in future revisions of this specification. Do not rely on OEM & vendor data starting at the same offset from the beginning of the IPXE structure. It is recommended that the OEM & vendor data include a signature that drivers/applications can search for.
### Status

This is a read-only port. Undefined status bits must be set to zero. Reading this port does NOT clear the status.

<table>
<thead>
<tr>
<th>Bit</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x00</td>
<td>Comm and completion interrupt pending (1) or not pending (0)</td>
</tr>
<tr>
<td>0x01</td>
<td>Packet received interrupt pending (1) or not pending (0)</td>
</tr>
<tr>
<td>0x02</td>
<td>Transmit complete interrupt pending (1) or not pending (0)</td>
</tr>
<tr>
<td>0x03</td>
<td>Software interrupt pending (1) or not pending (0)</td>
</tr>
<tr>
<td>0x04</td>
<td>Command completion interrupts enabled (1) or disabled (0)</td>
</tr>
<tr>
<td>0x05</td>
<td>Packet receive interrupts enabled (1) or disabled (0)</td>
</tr>
<tr>
<td>0x06</td>
<td>Transmit complete interrupts enabled (1) or disabled (0)</td>
</tr>
<tr>
<td>0x07</td>
<td>Software interrupts enabled (1) or disabled (0)</td>
</tr>
<tr>
<td>0x08</td>
<td>Unicast receive enabled (1) or disabled (0)</td>
</tr>
<tr>
<td>0x09</td>
<td>Filtered multicast receive enabled (1) or disabled (0)</td>
</tr>
<tr>
<td>0x0A</td>
<td>Broadcast receive enabled (1) or disabled (0)</td>
</tr>
<tr>
<td>0x0B</td>
<td>Promiscuous receive enabled (1) or disabled (0)</td>
</tr>
<tr>
<td>0x0C</td>
<td>Promiscuous multicast receive enabled (1) or disabled (0)</td>
</tr>
<tr>
<td>0x1D</td>
<td>Comm and failed (1) or command succeeded (0)</td>
</tr>
<tr>
<td>0x1F</td>
<td>UNDI state: Stopped (0), Started (1), Initialized (2), Busy (3)</td>
</tr>
</tbody>
</table>

### Command

Use to execute commands, clear interrupt status and enable/disable receive levels. This is a read/write port. Read reflects the last write.

<table>
<thead>
<tr>
<th>Bit</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x00</td>
<td>Clear command completion interrupt (1) or NOP (0)</td>
</tr>
<tr>
<td>0x01</td>
<td>Clear packet received interrupt (1) or NOP (0)</td>
</tr>
<tr>
<td>0x02</td>
<td>Clear transmit complete interrupt (1) or NOP (0)</td>
</tr>
<tr>
<td>0x03</td>
<td>Clear software interrupt (1) or NOP (0)</td>
</tr>
<tr>
<td>0x04</td>
<td>Command completion interrupt enabled (1) or disable (0)</td>
</tr>
<tr>
<td>0x05</td>
<td>Packet receive interrupt enable (1) or disable (0)</td>
</tr>
<tr>
<td>0x06</td>
<td>Transmit complete interrupt enable (1) or disable (0)</td>
</tr>
<tr>
<td>0x07</td>
<td>Software interrupt enable (1) or disable (0). Setting this bit to (1) also generates a software interrupt.</td>
</tr>
<tr>
<td>0x08</td>
<td>Unicast receive enable (1) or disable (0)</td>
</tr>
<tr>
<td>0x09</td>
<td>Filtered multicast receive enable (1) or disable (0)</td>
</tr>
<tr>
<td>0x0A</td>
<td>Broadcast receive enable (1) or disable (0)</td>
</tr>
<tr>
<td>0x0B</td>
<td>Promiscuous receive enable (1) or disable (0)</td>
</tr>
<tr>
<td>0x0C</td>
<td>Promiscuous multicast receive enable (1) or disable (0)</td>
</tr>
<tr>
<td>0x1F</td>
<td>Operation type: Clear interrupt and/or filter (0), Issue command (1)</td>
</tr>
</tbody>
</table>

### CDBaddr

Write the physical address of a CDB to this port. (Done with one 64-bit or two 32-bit writes, depending on processor architecture.) When done, use one 32-bit write to the command port to send this address into the command queue. Unused upper address bits must be set to zero.

### S/W UNDI Fields

- **EntryPoint**: S/W UNDI API entry point address. This is either a virtual address or an offset from the start of the IPXE structure. Protocol drivers will push the 64-bit virtual address of a CDB on the stack and then call the UNDI API entry point. When control is returned to the protocol driver, the protocol driver must remove the address of the CDB from the stack.

- **BusTypeCnt**: This field is the count of 4-byte BusType entries in the next field.

- **BusType**: This field defines the type of bus S/W UNDI is written to support: “PCIR,” “PCCR,” “USBR” or “1394.” This field is formatted like the Signature field. If the S/W UNDI supports more than one BusType there will be more than one BusType identifier in this field.
E.2.1.1 Issuing UNDI Commands

How commands are written and status is checked varies a little depending on the type of UNDI (H/W or S/W) implementation being used. The command flowchart shown in Figure E-3 is a high-level diagram on how commands are written to both H/W and S/W UNDI.

Figure E-3 Issuing UNDI Commands

E.2.2 UNDI Command Format

The format of the CDB is the same for all UNDI commands. Figure E-4 shows the structure of the CDB. Some of the commands do not use or always require the use of all of the fields in the CDB. When fields are not used they must be initialized to zero or the UNDI will return an error. The StatCode and StatFlags fields must always be initialized to zero or the UNDI will return an error. All reserved fields (and bit fields) must be initialized to zero or the UNDI will return an error.

Basically, the rule is: Do it right, or don’t do it at all.
Descriptions of the CDB fields are given in Table E-5.

Table E-5 UNDI CDB Field Definitions

<table>
<thead>
<tr>
<th>Identifier</th>
<th>Description</th>
</tr>
</thead>
</table>
| OpCode     | **Operation Code** (Function Number, Command Code, etc.)  
This field is used to identify the command being sent to the UNDI. The meanings of some of the  
bits in the OpFlags and StatFlags fields, and the format of the CPB and DB structures depends on  
the value in the OpCode field. Commands sent with an OpCode value that is not defined in this  
specification will not be executed and will return a StatCode of **PXE_STATCODE_INVALID_CDB**. |
| OpFlags    | Operation Flags  
This bit field is used to enable/disable different features in a specific command operation. It is  
also used to change the format/contents of the CPB and DB structures. Commands sent with  
reserved bits set in the OpFlags field will not be executed and will return a StatCode of  
**PXE_STATCODE_INVALID_CDB**. |
| CPBsize    | Command Parameter Block Size  
This field should be set to a number that is equal to the number of bytes that will be read from  
CPB structure during command execution. Setting this field to a number that is too small will  
cause the command to not be executed and a StatCode of **PXE_STATCODE_INVALID_CDB**  
will be returned.  
The contents of the CPB structure will not be modified. |
| DBsize     | Data Block Size  
This field should be set to a number that is equal to the number of bytes that will be written into  
the DB structure during command execution. Setting this field to a number that is smaller than  
required will cause an error. It may be zero in some cases where the information is not needed. |
<table>
<thead>
<tr>
<th>Identifier</th>
<th>Description</th>
</tr>
</thead>
</table>
| CPBaddr    | Command Parameter Block Address  
  For H/W UNDI, this field must be the physical address of the CPB structure. For S/W UNDI, this field must be the virtual address of the CPB structure. If the operation does not have/use a CPB, this field must be initialized to `PXE_CPBADDR_NOT_USED`. Setting up this field incorrectly will cause command execution to fail and a StatCode of `PXE_STATCODE_INVALID_CDB` will be returned. |
| DBaddr     | Data Block Address  
  For H/W UNDI, this field must be the physical address of the DB structure. For S/W UNDI, this field must be the virtual address of the DB structure. If the operation does not have/use a CPB, this field must be initialized to `PXE_DBADDR_NOT_USED`. Setting up this field incorrectly will cause command execution to fail and a StatCode of `PXE_STATCODE_INVALID_CDB` will be returned. |
| StatCode   | Status Code  
  This field is used to report the type of command completion: success or failure (and the type of failure). This field must be initialized to zero before the command is issued. The contents of this field is not valid until the `PXE_STATFLAGS_COMMAND_COMPLETE` status flag is set. If this field is not initialized to `PXE_STATCODE_INITIALIZE` the UNDI command will not execute and a StatCode of `PXE_STATCODE_INVALID_CDB` will be returned. |
| StatFlags  | Status Flags  
  This bit field is used to report command completion and identify the format, if any, of the DB structure. This field must be initialized to zero before the command is issued. Until the command state changes to error or complete, all other CDB fields must not be changed. If this field is not initialized to `PXE_STATFLAGS_INITIALIZE` the UNDI command will not execute and a StatCode of `PXE_STATCODE_INVALID_CDB` will be returned.  
  Bits 0x0F & 0x0E: Command state: Not started (0), Queued (1), Error (2), Complete (3). |
| IFnum      | Interface Number  
  This field is used to identify which network adapter (S/W UNDI) or network connector (H/W UNDI) this command is being sent to. If an invalid interface number is given, the command will not execute and a StatCode of `PXE_STATCODE_INVALID_CDB` will be returned. |
| Control    | Process Control  
  This bit field is used to control command UNDI inter-command processing. Setting control bits that are not supported by the UNDI will cause the command execution to fail with a StatCode of `PXE_STATCODE_INVALID_CDB`.  
  Bit 0x00: Another CDB follows this one (1) or this is the last or only CDB in the list (0).  
  Bit 0x01: Queue command if busy (1), fail if busy (0). |
E.3 UNDI C Definitions

The definitions in this section are used to aid in the portability and readability of the example 32/64-bit S/W UNDI source code and the rest of this specification.

E.3.1 Portability Macros

These macros are used for storage and communication portability.

E.3.1.1 PXE_INTEL_ORDER or PXE_NETWORK_ORDER

This macro is used to control conditional compilation in the S/W UNDI source code. One of these definitions needs to be uncommented in a common PXE header file.

```c
//#define PXE_INTEL_ORDER 1 // little-endian
//#define PXE_NETWORK_ORDER 1 // big-endian
```

E.3.1.2 PXE_UINT64_SUPPORT or PXE_NO_UINT64_SUPPORT

This macro is used to control conditional compilation in the PXE source code. One of these definitions must to be uncommented in the common PXE header file.

```c
//#define PXE_UINT64_SUPPORT 1 // UINT64 supported
//#define PXE_NO_UINT64_SUPPORT 1 // UINT64 not supported
```

E.3.1.3 PXE_BUSTYPE

Used to convert a 4-character ASCII identifier to a 32-bit unsigned integer.

```c
#if PXE_INTEL_ORDER
#define PXE_BUSTYPE(a,b,c,d)     
(((PXE_UINT32)(d) & 0xFF) << 24) | 
(((PXE_UINT32)(c) & 0xFF) << 16) | 
(((PXE_UINT32)(b) & 0xFF) << 8) | 
((PXE_UINT32)(a) & 0xFF))
#else
#define PXE_BUSTYPE(a,b,c,d)     
(((PXE_UINT32)(a) & 0xFF) << 24) | 
(((PXE_UINT32)(b) & 0xFF) << 16) | 
(((PXE_UINT32)(c) & 0xFF) << 8) | 
((PXE_UINT32)(f) & 0xFF))
#endif
```

//*******************************************************
// UNDI ROM ID and device ID signature
//*******************************************************
#define PXE_BUSTYPE_PXE PXE_BUSTYPE('!', 'P', 'X', 'E')

//*******************************************************
// BUS ROM ID signatures
//*******************************************************
#define PXE_BUSTYPE_PCI PXE_BUSTYPE('P', 'C', 'I', 'R')
#define PXE_BUSTYPE_PC_CARD PXE_BUSTYPE('P', 'C', 'C', 'R')
#define PXE_BUSTYPE_USB PXE_BUSTYPE('U', 'S', 'B', 'R')
#define PXE_BUSTYPE_1394 PXE_BUSTYPE('1', '3', '9', '4')

E.3.1.4 PXE_SWAP_UINT16

This macro swaps bytes in a 16-bit word.

#ifdef PXE_INTEL_ORDER
#define PXE_SWAP_UINT16(n)   
((((PXE_UINT16)(n) & 0x00FF) << 8) | 
(((PXE_UINT16)(n) & 0xFF00) >> 8))
#else
#define PXE_SWAP_UINT16(n) (n)
#endif

E.3.1.5 PXE_SWAP_UINT32

This macro swaps bytes in a 32-bit word.

#ifdef PXE_INTEL_ORDER
#define PXE_SWAP_UINT32(n)         
((((PXE_UINT32)(n) & 0x000000FF) << 24) | 
(((PXE_UINT32)(n) & 0x0000FF00) << 8) |  
(((PXE_UINT32)(n) & 0x00FF0000) >> 8) |  
(((PXE_UINT32)(n) & 0xFF000000) >> 24))
#else
#define PXE_SWAP_UINT32(n)   (n)
#endif

E.3.1.6 PXE_SWAP_UINT64

This macro swaps bytes in a 64-bit word for compilers that support 64-bit words.

#if PXE_UINT64_SUPPORT != 0
#ifdef PXE_INTEL_ORDER
#define PXE_SWAP_UINT64(n)     
((((PXE_UINT64)(n) & 0x00000000000000FF) << 56) | 
(((PXE_UINT64)(n) & 0x000000000000FF00) << 40) |  
(((PXE_UINT64)(n) & 0x0000000000FF0000) << 24) |  
(((PXE_UINT64)(n) & 0x00000000FF000000) << 8) |  
(((PXE_UINT64)(n) & 0x000000FF00000000) >> 8) |  
(((PXE_UINT64)(n) & 0x0000FF0000000000) >> 24) |  
(((PXE_UINT64)(n) & 0x00FF000000000000) >> 40) | 
(((PXE_UINT64)(n) & 0xFF00000000000000) >> 56))
#else
#define PXE_SWAP_UINT64(n) (n)
#endif
#endif // PXE_UINT64_SUPPORT
This macro swaps bytes in a 64-bit word, in place, for compilers that do not support 64-bit words. This version of the 64-bit swap macro cannot be used in expressions.

```c
#if PXE_NO_UINT64_SUPPORT != 0
#if PXE_INTEL.Order
#define PXE_SWAP_UINT64(n)  
{                       
  PXE_UINT32 tmp = (PXE_UINT64)(n)[1];      
  (PXE_UINT64)(n)[1] = PXE_SWAP_UINT32((PXE_UINT64)(n)[0]); 
  (PXE_UINT64)(n)[0] = PXE_SWAP_UINT32(tmp);    
}         
#else
#define PXE_SWAP_UINT64(n) (n)
#endif
#endif // PXE_NO_UINT64_SUPPORT
```

E.3.2 Miscellaneous Macros

E.3.2.1 Miscellaneous

```
#define PXE_CPBSIZE_NOT_USED  0   // zero
#define PXE_DBSIZE_NOT_USED  0   // zero
#define PXE_CPBADDR_NOT_USED  (PXE_UINT64)0 // zero
#define PXE_DBADDR_NOT_USED  (PXE_UINT64)0 // zero
```

E.3.3 Portability Types

The examples given below are just that, examples. The actual typedef instructions used in a new implementation may vary depending on the compiler and processor architecture.

The storage sizes defined in this section are critical for PXE module inter-operation. All of the portability typedefs define little endian (Intel® format) storage. The least significant byte is stored in the lowest memory address and the most significant byte is stored in the highest memory address, as shown in Figure E-5.

![Figure E-5 Storage Types](image-url)
E.3.3.1 PXE_CONST

The const type does not allocate storage. This type is a modifier that is used to help the compiler optimize parameters that do not change across function calls.

#define PXE_CONST const

E.3.3.2 PXE_VOLATILE

The volatile type does not allocate storage. This type is a modifier that is used to help the compiler deal with variables that can be changed by external procedures or hardware events.

#define PXE_VOLATILE volatile

E.3.3.3 PXE_VOID

The void type does not allocate storage. This type is used only to prototype functions that do not return any information and/or do not take any parameters.

typedef void PXE_VOID;

E.3.3.4 PXE_UINT8

Unsigned 8-bit integer.

typedef unsigned char PXE_UINT8;

E.3.3.5 PXE_UINT16

Unsigned 16-bit integer.

typedef unsigned short PXE_UINT16;

E.3.3.6 PXE_UINT32

Unsigned 32-bit integer.

typedef unsigned PXE_UINT32;

E.3.3.7 PXE_UINT64

Unsigned 64-bit integer.

#if PXE_UINT64_SUPPORT != 0
typedef unsigned long PXE_UINT64;
#endif // PXE_UINT64_SUPPORT

If a 64-bit integer type is not available in the compiler being used, use this definition:

#if PXE_NO_UINT64_SUPPORT != 0
typedef PXE_UINT32 PXE_UINT64[2];
#endif // PXE_NO_UINT64_SUPPORT
E.3.3.8 PXE_UINTN

Unsigned integer that is the default word size used by the compiler. This needs to be at least a 32-bit unsigned integer.

```c
typedef unsigned PXE_UINTN;
```

E.3.4 Simple Types

The PXE simple types are defined using one of the portability types from the previous section.

E.3.4.1 PXE_BOOL

Boolean (true/false) data type. For PXE zero is always false and nonzero is always true.

```c
typedef PXE_UINT8 PXE_BOOL;
#define PXE_FALSE 0 // zero
#define PXE_TRUE (!PXE_FALSE)
```

E.3.4.2 PXE_OPCODE

UNDI OpCode (command) descriptions are given in the next chapter. There are no BC OpCodes, BC protocol functions are discussed later in this document.

```c
typedef PXE_UINT16 PXE_OPCODE;

// Return UNDI operational state.
#define PXE_OPCODE_GET_STATE 0x0000

// Change UNDI operational state from Stopped to Started.
#define PXE_OPCODE_START 0x0001

// Change UNDI operational state from Started to Stopped.
#define PXE_OPCODE_STOP 0x0002

// Get UNDI initialization information.
#define PXE_OPCODE_GET_INIT_INFO 0x0003

// Get NIC configuration information.
#define PXE_OPCODE_GET_CONFIG_INFO 0x0004

// Changed UNDI operational state from Started to Initialized.
#define PXE_OPCODE_INITIALIZE 0x0005

// Reinitialize the NIC H/W.
#define PXE_OPCODE_RESET 0x0006

// Change the UNDI operational state from Initialized to Started.
#define PXE_OPCODE_SHUTDOWN 0x0007
```
// Read & change state of external interrupt enables.
#define PXE_OPCODE_INTERRUPT_ENABLES   0x0008

// Read & change state of packet receive filters.
#define PXE_OPCODE_RECEIVE_FILTERS   0x0009

// Read & change station MAC address.
#define PXE_OPCODE_STATION_ADDRESS   0x000A

// Read traffic statistics.
#define PXE_OPCODE_STATISTICS    0x000B

// Convert multicast IP address to multicast MAC address.
#define PXE_OPCODE_MCAST_IP_TO_MAC   0x000C

// Read or change nonvolatile storage on the NIC.
#define PXE_OPCODE_NVDATA     0x000D

// Get & clear interrupt status.
#define PXE_OPCODE_GET_STATUS    0x000E

// Fill media header in packet for transmit.
#define PXE_OPCODE_FILL_HEADER    0x000F

// Transmit packet(s).
#define PXE_OPCODE_TRANSMIT     0x0010

// Receive packet.
#define PXE_OPCODE_RECEIVE     0x0011

// Last valid PXE UNDI OpCode number.
#define PXE_OPCODE_LAST_VALID    0x0011

E.3.4.3 PXE_OPFLAGS

typedef PXE_UINT16 PXE_OPFLAGS;

#define PXE_OPFLAGS_NOT_USED    0x0000

// UNDI Get State

// No OpFlags

// UNDI Start
// No OpFlags

// UNDI Stop
// No OpFlags

// UNDI Get Init Info
// No Opflags

// UNDI Get Config Info
// No Opflags

// UNDI Initialize
//*******************************************************************************

#define PXE_OPFLAGS_INITIALIZE_CABLE_DETECT_MASK  0x0001
#define PXE_OPFLAGS_INITIALIZE_DETECT_CABLE   0x0000
#define PXE_OPFLAGS_INITIALIZE_DO_NOT_DETECT_CABLE 0x0001

//*******************************************************************************
// UNDI Reset
//*******************************************************************************

#define PXE_OPFLAGS_RESET_DISABLE_INTERRUPTS  0x0001
#define PXE_OPFLAGS_RESET_DISABLE_FILTERS    0x0002

//*******************************************************************************
// UNDI Shutdown
//*******************************************************************************

// No OpFlags

//*******************************************************************************
// UNDI Interrupt Enables
//*******************************************************************************
// Select whether to enable or disable external interrupt signals. Setting both enable and disable will return PXE_STATCODE_INVALID_OPFLAGS.

#define PXE_OPFLAGS_INTERRUPT_OPMASK    0xC000
#define PXE_OPFLAGS_INTERRUPT_ENABLE    0x8000
#define PXE_OPFLAGS_INTERRUPT_DISABLE    0x4000
#define PXE_OPFLAGS_INTERRUPT_READ     0x0000

// Enable receive interrupts. An external interrupt will be generated after a complete non-error packet has been received.

#define PXE_OPFLAGS_INTERRUPT_RECEIVE    0x0001

// Enable transmit interrupts. An external interrupt will be generated after a complete non-error packet has been transmitted.

#define PXE_OPFLAGS_INTERRUPT_TRANSMIT    0x0002

// Enable command interrupts. An external interrupt will be generated when command execution stops.

#define PXE_OPFLAGS_INTERRUPT_COMMAND    0x0004

// Generate software interrupt. Setting this bit generates an external interrupt, if it is supported by the hardware.

#define PXE_OPFLAGS_INTERRUPT_SOFTWARE    0x0008

UNDI Receive Filters

// Select whether to enable or disable receive filters. Setting both enable and disable will return PXE_STATCODE_INVALID_OPCODE.

#define PXE_OPFLAGS_RECEIVE_FILTER_OPMASK   0xC000
#define PXE_OPFLAGS_RECEIVE_FILTER_ENABLE   0x8000
#define PXE_OPFLAGS_RECEIVE_FILTER_DISABLE   0x4000
#define PXE_OPFLAGS_RECEIVE_FILTER_READ    0x0000

// To reset the contents of the multicast MAC address filter list, set this OpFlag:

#define PXE_OPFLAGS_RECEIVE_FILTERS_RESET_MCAST_LIST 0x2000
// Enable unicast packet receiving. Packets sent to the
// current station MAC address will be received.
#define PXE_OPFLAGS_RECEIVE_FILTER_UNICAST 0x0001

// Enable broadcast packet receiving. Packets sent to the
// broadcast MAC address will be received.
#define PXE_OPFLAGS_RECEIVE_FILTER_BROADCAST 0x0002

// Enable filtered multicast packet receiving. Packets sent to
// any of the multicast MAC addresses in the multicast MAC
// address filter list will be received. If the filter list is
// empty, no multicast
#define PXE_OPFLAGS_RECEIVE_FILTER_FILTERED_MULTICAST 0x0004

// Enable promiscuous packet receiving. All packets will be
// received.
#define PXE_OPFLAGS_RECEIVE_FILTER_PROMISCUOUS 0x0008

// Enable promiscuous multicast packet receiving. All multicast
// packets will be received.
#define PXE_OPFLAGS_RECEIVE_FILTER_ALL_MULTICAST 0x0010

// UNDI Station Address

// UNDI Station Address
#define PXE_OPFLAGS_STATION_ADDRESS_READ 0x0000
#define PXE_OPFLAGS_STATION_ADDRESS_WRITE 0x0000
#define PXE_OPFLAGS_STATION_ADDRESS_RESET 0x0001

// UNDI Statistics

// UNDI MCast IP to MAC

// Identify the type of IP address in the CPB.
```c
#define PXE_OPFLAGS_MCAST_IP_TO_MAC_OPMASK 0x0003
#define PXE_OPFLAGS_MCAST_IPV4_TO_MAC 0x0000
#define PXE_OPFLAGS_MCAST_IPV6_TO_MAC 0x0001

//UNDI NvData

// Select the type of nonvolatile data operation.
#define PXE_OPFLAGS_NVDATA_OPMASK 0x0001
#define PXE_OPFLAGS_NVDATA_READ 0x0000
#define PXE_OPFLAGS_NVDATA_WRITE 0x0001

//UNDI Get Status

// Return current interrupt status. This will also clear any interrupts that are currently set. This can be used in a polling routine. The interrupt flags are still set and cleared even when the interrupts are disabled.
#define PXE_OPFLAGS_GET_INTERRUPT_STATUS 0x0001

// Return list of transmitted buffers for recycling. Transmit buffers must not be changed or unallocated until they have recycled. After issuing a transmit command, wait for a transmit complete interrupt. When a transmit complete interrupt is received, read the transmitted buffers. Do not plan on getting one buffer per interrupt. Some NICs and UNDIs may transmit multiple buffers per interrupt.
#define PXE_OPFLAGS_GET_TRANSMITTED_BUFFERS 0x0002

// Return current media status.
#define PXE_OPFLAGS_GET_MEDIA_STATUS 0x0004

//UNDI Fill Header

// Define flags for fill header operation.
#define PXE_OPFLAGS_FILL_HEADER_OPMASK 0x0001
#define PXE_OPFLAGS_FILL_HEADER_FRAGMENTED 0x0001
#define PXE_OPFLAGS_FILL_HEADER_WHOLE 0x0000
```
//*******************************************************
// UNDI Transmit
//@******************************************************************************

// S/W UNDI only. Return after the packet has been transmitted.
// A transmit complete interrupt will still be generated and the
// transmit buffer will have to be recycled.

#define PXE_OPFLAGS_SWUNDI_TRANSMIT_OPMASK 0x0001
#define PXE_OPFLAGS_TRANSMIT_BLOCK 0x0001
#define PXE_OPFLAGS_TRANSMIT_DONT_BLOCK 0x0000

#define PXE_OPFLAGS_TRANSMIT_OPMASK 0x0002
#define PXE_OPFLAGS_TRANSMIT_FRAGMENTED 0x0002
#define PXE_OPFLAGS_TRANSMIT_WHOLE 0x0000
//@******************************************************************************

// UNDI Receive
//@******************************************************************************

//@******************************************************************************
// No OpFlags

E.3.4.4 PXE_STATFLAGS
//@******************************************************************************

typedef PXE_UINT16 PXE_STATFLAGS;

#define PXE_STATFLAGS_INITIALIZE 0x0000
//@******************************************************************************

//@******************************************************************************
// Common StatFlags that can be returned by all commands.
//@******************************************************************************

//@******************************************************************************
// The COMMAND_COMPLETE and COMMAND_FAILED status flags must be
// implemented by all UNDIs. COMMAND_QUEUED is only needed by
// UNDIs that support command queuing.
//@******************************************************************************

#define PXE_STATFLAGS_STATUS_MASK 0xC000
#define PXE_STATFLAGS_COMMAND_COMPLETE 0xC000
#define PXE_STATFLAGS_COMMAND_FAILED 0x8000
#define PXE_STATFLAGS_COMMAND_QUEUED 0x4000
//******************************************************************************
// UNDI Get State
//******************************************************************************
#define PXE_STATFLAGS_GET_STATE_MASK 0x0003
#define PXE_STATFLAGS_GET_STATE_INITIALIZED 0x0002
#define PXE_STATFLAGS_GET_STATE_STARTED 0x0001
#define PXE_STATFLAGS_GET_STATE_STOPPED 0x0000

//******************************************************************************
// UNDI Start
//******************************************************************************
// No additional StatFlags

//******************************************************************************
// UNDI Get Init Info
//******************************************************************************
#define PXE_STATFLAGS_CABLE_DETECT_MASK 0x0001
#define PXE_STATFLAGS_CABLE_DETECT_NOT_SUPPORTED 0x0000
#define PXE_STATFLAGS_CABLE_DETECT_SUPPORTED 0x0001
#define PXE_STATFLAGS_GET_STATUS_NO_MEDIA_MASK 0x0002
#define PXE_STATFLAGS_GET_STATUS_NO_MEDIA_NOT_SUPPORTED 0x0000
#define PXE_STATFLAGS_GET_STATUS_NO_MEDIA_SUPPORTED 0x0002

//******************************************************************************
// UNDI Initialize
//******************************************************************************
#define PXE_STATFLAGS_INITIALIZED_NO_MEDIA 0x0001

//******************************************************************************
// UNDI Reset
//******************************************************************************
#define PXE_STATFLAGS_RESET_NO_MEDIA 0x0001

//******************************************************************************
// UNDI Shutdown
//******************************************************************************
// No additional StatFlags
// UNDI Interrupt Enables

// If set, receive interrupts are enabled.
#define PXE_STATFLAGS_INTERRUPT_RECEIVE  0x0001

// If set, transmit interrupts are enabled.
#define PXE_STATFLAGS_INTERRUPT_TRANSMIT  0x0002

// If set, command interrupts are enabled.
#define PXE_STATFLAGS_INTERRUPT_COMMAND   0x0004

// UNDI Receive Filters

// If set, unicast packets will be received.
#define PXE_STATFLAGS_RECEIVE_FILTER_UNICAST  0x0001

// If set, broadcast packets will be received.
#define PXE_STATFLAGS_RECEIVE_FILTER_BROADCAST  0x0002

// If set, multicast packets that match up with the multicast
// address filter list will be received.
#define PXE_STATFLAGS_RECEIVE_FILTER_FILTERED_MULTICAST 0x0004

// If set, all packets will be received.
#define PXE_STATFLAGS_RECEIVE_FILTER_PROMISCUOUS  0x0008

// If set, all multicast packets will be received.
#define PXE_STATFLAGS_RECEIVE_FILTER_ALL_MULTICAST 0x0010

// UNDI Station Address

// No additional StatFlags

// UNDI Statistics

// No additional StatFlags
// UNDI MCast IP to MAC
//*******************************************************

// No additional StatFlags
//*******************************************************

// UNDI NvData
//*******************************************************

// No additional StatFlags
//*******************************************************

// UNDI Get Status
//*******************************************************

// Use to determine if an interrupt has occurred.
#define PXE_STATFLAGS_GET_STATUS_INTERRUPT_MASK 0x000F
#define PXE_STATFLAGS_GET_STATUS_NO_INTERRUPTS 0x0000

// If set, at least one receive interrupt occurred.
#define PXE_STATFLAGS_GET_STATUS_RECEIVE 0x0001

// If set, at least one transmit interrupt occurred.
#define PXE_STATFLAGS_GET_STATUS_TRANSMIT 0x0002

// If set, at least one command interrupt occurred.
#define PXE_STATFLAGS_GET_STATUS_COMMAND 0x0004

// If set, at least one software interrupt occurred.
#define PXE_STATFLAGS_GET_STATUS_SOFTWARE 0x0008

// This flag is set if the transmitted buffer queue is empty.
// This flag will be set if all transmitted buffer addresses
// get written into the DB.
#define PXE_STATFLAGS_GET_STATUS_TXBUF_QUEUE_EMPTY 0x0010

// This flag is set if no transmitted buffer addresses were
// written into the DB. (This could be because DBsize was
// too small.)
#define PXE_STATFLAGS_GET_STATUS_NO_TXBUFS_WRITTEN 0x0020

// This flag is set if there is no media detected
#define PXE_STATFLAGS_GET_STATUS_NO_MEDIA 0x0040

// UNDI Fill Header
E.3.4.5 PXE_STATCODE

typedef PXE_UINT16 PXE_STATCODE;

#define PXE_STATCODE_INITIALIZE     0x0000

#define PXE_STATCODE_SUCCESS        0x0000
#define PXE_STATCODE_INVALID_CDB      0x0001
#define PXE_STATCODE_INVALID_CPB      0x0002
#define PXE_STATCODE_BUSY         0x0003
#define PXE_STATCODE_QUEUE_FULL      0x0004
#define PXE_STATCODE_ALREADY_STARTED    0x0005
#define PXE_STATCODE_NOT_STARTED      0x0006
#define PXE_STATCODE_NOT_SHUTDOWN     0x0007
#define PXE_STATCODE_ALREADY_INITIALIZED  0x0008
#define PXE_STATCODE_NOT_INITIALIZED    0x0009
#define PXE_STATCODE_DEVICE_FAILURE    0x000A
#define PXE_STATCODE_NVDATA_FAILURE    0x000B
#define PXE_STATCODE_UNSUPPORTED      0x000C
#define PXE_STATCODE_BUFFER_FULL      0x000D
#define PXE_STATCODE_INVALID_PARAMETER   0x000E
#define PXE_STATCODE_INVALID_UNDI     0x000F
#define PXE_STATCODE_IPV4_NOT_SUPPORTED  0x0010
#define PXE_STATCODE_IPV6_NOT_SUPPORTED  0x0011
#define PXE_STATCODE_NOT_ENOUGH_MEMORY   0x0012
#define PXE_STATCODE_NO_DATA        0x0013
E.3.4.6 PXE_IFNUM

typedef PXE_UINT16 PXE_IFNUM;

// This interface number must be passed to the S/W UNDI Start // command.

#define PXE_IFNUM_START 0x0000

// This interface number is returned by the S/W UNDI Get State // and Start commands if information in the CDB, CPB or DB is // invalid.

#define PXE_IFNUM_INVALID 0x0000

E.3.4.7 PXE_CONTROL

typedef PXE_UINT16 PXE_CONTROL;

// Setting this flag directs the UNDI to queue this command for // later execution if the UNDI is busy and it supports command // queuing. If queuing is not supported, a // PXE_STATCODE_INVALID_CONTROL error is returned. If the queue // is full, a PXE_STATCODE_CDB_QUEUE_FULL error is returned.

#define PXE_CONTROL_QUEUE_IF_BUSY 0x0002

// These two bit values are used to determine if there are more // UNDI CDB structures following this one. If the link bit is // set, there must be a CDB structure following this one. // Execution will start on the next CDB structure as soon as this // one completes successfully. If an error is generated by this // command, execution will stop.

#define PXE_CONTROL_LINK 0x0001
#define PXE_CONTROL_LAST_CDB_IN_LIST 0x0000

E.3.4.8 PXE_FRAME_TYPE

typedef PXE_UINT8 PXE_FRAME_TYPE;

#define PXE_FRAME_TYPE_NONE 0x00
#define PXE_FRAME_TYPE_UNICAST 0x01
#define PXE_FRAME_TYPE_BROADCAST 0x02
#define PXE_FRAME_TYPE_FILTERED_MULTICAST 0x03
#define PXE_FRAME_TYPE_PROMISCUOUS 0x04
#define PXE_FRAME_TYPE_PROMISCUOUS_MULTICAST 0x05
E.3.4.9 PXE_IPV4
This storage type is always big endian, not little endian.

```c
typedef PXE_UINT32 PXE_IPV4;
```

E.3.4.10 PXE_IPV6
This storage type is always big endian, not little endian.

```c
typedef struct s_PXE_IPV6 {
    PXE_UINT32 num[4];
} PXE_IPV6;
```

E.3.4.11 PXE_MAC_ADDR
This storage type is always big endian, not little endian.

```c
typedef struct {
    PXE_UINT8 num[32];
} PXE_MAC_ADDR;
```

E.3.4.12 PXE_IFTYPE
The interface type is returned by the Get Initialization Information command and is used by the BC DHCP protocol function. This field is also used for the low order 8-bits of the H/W type field in ARP packets. The high order 8-bits of the H/W type field in ARP packets will always be set to 0x00 by the BC.

```c
typedef PXE_UINT8 PXE_IFTYPE;
```

// This information is from the ARP section of RFC 3232.

```c
//  1 Ethernet (10Mb)
//  2 Experimental Ethernet (3Mb)
//  3 Amateur Radio AX.25
//  4 Proteon ProNET Token Ring
//  5 Chaos
//  6 IEEE 802 Networks
//  7 ARCNET
//  8 Hyperchannel
//  9 Lanstar
// 10 Autonet Short Address
// 11 LocalTalk
// 12 LocalNet (IBM PCNet or SYTEK LocalNET)
// 13 Ultra link
// 14 SMDS
// 15 Frame Relay
// 16 Asynchronous Transmission Mode (ATM)
// 17 HDLC
// 18 Fibre Channel
// 19 Asynchronous Transmission Mode (ATM)
```
// 20 Serial Line
// 21 Asynchronous Transmission Mode (ATM)

#define PXE_IFTYPE_ETHERNET     0x01
#define PXE_IFTYPE_TOKENRING    0x04
#define PXE_IFTYPE_FIBRE_CHANNEL 0x12

E.3.4.13 PXE_MEDIA_PROTOCOL
Protocol type. This will be copied into the media header without doing byte swapping. Protocol type numbers can be obtained from the assigned numbers RFC 3232.

typedef UINT16   PXE_MEDIA_PROTOCOL;

E.3.5 Compound Types
All PXE structures must be byte packed.

E.3.5.1 PXE_HW_UNDI
This section defines the C structures and #defines for the PXE H/W UNDI interface.

#pragma pack(1)
typedef struct s_pxe_hw_undi {
    PXE_UINT32   Signature;  // PXE_ROMID_SIGNATURE
    PXE_UINT8    Len;        // sizeof(PXE_HW_UNDI)
    PXE_UINT8    Fudge;      // makes 8-bit checksum equal zero
    PXE_UINT8    Rev;        // PXE_ROMID_REV
    PXE_UINT8    IFcnt;      // physical connector count
        lower byte
    PXE_UINT8    MajorVer;   // PXE_ROMID_MAJORVER
    PXE_UINT8    MinorVer;   // PXE_ROMID_MINORVER
    PXE_UINT8    IFcntExt;   // physical connector count
        upper byte
    PXE_UINT8    reserved;   // zero, not used
    PXE_UINT32   Implementation; // implementation flags
} PXE_HW_UNDI;
#pragma pack()

// Status port bit definitions

// UNDI operation state

#define PXE_HWSSTAT_STATE_MASK     0xC0000000
#define PXE_HWSSTAT_BUSY           0xC0000000
#define PXE_HWSSTAT_INITIALIZED    0x80000000
#define PXE_HWSSTAT_STARTED        0x40000000
#define PXE_HWSSTAT_STOPPED        0x00000000
// If set, last command failed
#define PXE_HWSTAT_COMMAND_FAILED 0x20000000

// If set, identifies enabled receive filters
#define PXE_HWSTAT_PROMISCUOUS_MULTICAST_RX_ENABLED 0x00001000
#define PXE_HWSTAT_MULTICAST_RX_ENABLED 0x00000200
#define PXE_HWSTAT_UNICAST_RX_ENABLED 0x00000100

// If set, identifies enabled external interrupts
#define PXE_HWSTAT_SOFTWARE_INT_ENABLED 0x00000080
#define PXE_HWSTAT_TX_COMPLETE_INT_ENABLED 0x00000040
#define PXE_HWSTAT_PACKET_RX_INT_ENABLED 0x00000020
#define PXE_HWSTAT_CMD_COMPLETE_INT_ENABLED 0x00000010

// If set, identifies pending interrupts
#define PXE_HWSTAT_SOFTWARE_INT_PENDING 0x00000008
#define PXE_HWSTAT_TX_COMPLETE_INT_PENDING 0x00000004
#define PXE_HWSTAT_PACKET_RX_INT_PENDING 0x00000002
#define PXE_HWSTAT_CMD_COMPLETE_INT_PENDING 0x00000001

// Command port definitions

// If set, CDB identified in CDBaddr port is given to UNDI.  
// If not set, other bits in this word will be processed.
#define PXE_HWCMD_ISSUE_COMMAND 0x80000000
#define PXE_HWCMD_INTS_AND_FILTS 0x00000000

// Use these to enable/disable receive filters.
#define PXE_HWCMD_PROMISCUOUS_MULTICAST_RX_ENABLE 0x00001000
#define PXE_HWCMD_MULTICAST_RX_ENABLE 0x00000200
#define PXE_HWCMD_UNICAST_RX_ENABLE 0x00000100

// Use these to enable/disable external interrupts
#define PXE_HWCMD_SOFTWARE_INT_ENABLE 0x00000080
#define PXE_HWCMD_TX_COMPLETE_INT_ENABLE 0x00000040
#define PXE_HWCMD_PACKET_RX_INT_ENABLE 0x00000020
#define PXE_HWCMD_CMD_COMPLETE_INT_ENABLE  0x00000010

// Use these to clear pending external interrupts
#define PXE_HWCMD_CLEAR_SOFTWARE_INT   0x00000008
#define PXE_HWCMD_CLEAR_TX_COMPLETE_INT   0x00000004
#define PXE_HWCMD_CLEAR_PACKET_RX_INT    0x00000002
#define PXE_HWCMD_CLEAR_CMD_COMPLETE_INT   0x00000001

E.3.5.2 PXE_SW_UNDI

This section defines the C structures and #defines for the !PXE S/W UNDI interface.

#pragma pack(1)
typedef struct s_pxe_sw_undi {
    PXE_UINT32 Signature;  // PXE_ROMID_SIGNATURE
    PXE_UINT8 Len;  // sizeof(PXE_SW_UNDI)
    PXE_UINT8 Fudge;  // makes 8-bit checksum zero
    PXE_UINT8 Rev;  // PXE_ROMID_REV
    PXE_UINT8 IFcnt;  // physical connector count
    lower byte
    PXE_UINT8 MajorVer;  // PXE_ROMID_MAJORVER
    PXE_UINT8 MinorVer;  // PXE_ROMID_MINORVER
    PXE_UINT8 IFcntExt;  // physical connector count
    upper byte
    PXE_UINT8 reserved1;  // zero, not used
    PXE_UINT32 Implementation;  // Implementation flags
    PXE_UINT64 EntryPoint;  // API entry point
    PXE_UINT8 reserved2[3];  // zero, not used
    PXE_UINT8 BusCnt;  // number of bustypes supported
    PXE_UINT32 BusType[1];  // list of supported bustypes
} PXE_SW_UNDI;
#pragma pack()

E.3.5.3 PXE_UNDI

PXE_UNDI combines both the H/W and S/W UNDI types into one typedef and has #defines for common fields in both H/W and S/W UNDI types.

#pragma pack(1)
typedef union u_pxe_undi {
    PXE_HW_UNDI hw;
    PXE_SW_UNDI sw;
} PXE_UNDI;
#pragma pack()
#define PXE_ROMID_SIGNATURE PXE_BUSTYPE ('!', 'P', 'X', 'E')

// !PXE structure format revision )
// See “Links to UEFI-Related Documents” (http://uefi.org/uefi)
// under the heading “UDP over IPv6”.

#define PXE_ROMID_REV 0x02

// UNDI command interface revision. These are the values that
// get sent in option 94 (Client Network Interface Identifier) in
// the DHCP Discover and PXE Boot Server Request packets.
// See “Links to UEFI-Related Documents” (http://uefi.org/uefi)
// under the heading “IETF Organization”.

#define PXE_ROMID_MAJORVER 0x03
#define PXE_ROMID_MINORVER 0x01

// Implementation flags

#define PXE_ROMID_IMP_HW_UNDI 0x80000000
#define PXE_ROMID_IMP_SW_VIRT_ADDR 0x40000000
#define PXE_ROMID_IMP_64BIT_DEVICE 0x00010000
#define PXE_ROMID_IMP_FRAG_SUPPORTED 0x00008000
#define PXE_ROMID_IMP_CMD_LINK_SUPPORTED 0x00004000
#define PXE_ROMID_IMP_CMD_QUEUE_SUPPORTED 0x00002000
#define PXE_ROMID_IMP_MULTI_FRAME_SUPPORTED 0x00001000
#define PXE_ROMID_IMP_NVDATA_SUPPORT_MASK 0x00000C00
#define PXE_ROMID_IMP_NVDATA_BULK_WRITABLE 0x00000C00
#define PXE_ROMID_IMP_NVDATA_SPARSE_WRITABLE 0x00000800
#define PXE_ROMID_IMP_NVDATA_READ_ONLY 0x00000400
#define PXE_ROMID_IMP_NVDATA_NOT_AVAILABLE 0x00000000
#define PXE_ROMID_IMP_STATISTICS_SUPPORTED 0x00000200
#define PXE_ROMID_IMP_STATION_ADDR_SETTABLE 0x00000100
#define PXE_ROMID_IMP_PROMISCUOUS_MULTICAST_RX_SUPPORTED 0x00000080
#define PXE_ROMID_IMP_PROMISCUOUS_RX_SUPPORTED 0x00000040
#define PXE_ROMID_IMP_BROADCAST_RX_SUPPORTED 0x00000020
#define PXE_ROMID_IMP_FILTERED_MULTICAST_RX_SUPPORTED 0x00000010
#define PXE_ROMID_IMP_SOFTWARE_INT_SUPPORTED 0x00000008
#define PXE_ROMID_IMP_TX_COMPLETE_INT_SUPPORTED 0x00000004
#define PXE_ROMID_IMP_PACKET_RX_INT_SUPPORTED 0x00000002
#define PXE_ROMID_IMP_CMD_COMPLETE_INT_SUPPORTED 0x00000001
E.3.5.4 PXE_CDB

PXE UNDI command descriptor block.

```c
#pragma pack(1)
typedef struct s_pxe_cdb {
    PXE_OPCODE     OpCode;
    PXE_OPFLAGS    OpFlags;
    PXE_UINT16     CPBsize;
    PXE_UINT16     DBsize;
    PXE_UINT64     CPBaddr;
    PXE_UINT64     DBaddr;
    PXE_STATCODE   StatCode;
    PXE_STATFLAGS  StatFlags;
    PXE_UINT16     IFnum;
    PXE_CONTROL    Control;
} PXE_CDB;
#pragma pack()
```

If the UNDI driver enables hardware VLAN support, UNDI driver could use `IFnum` to identify the real NICs and VLAN created virtual NICs.

E.3.5.5 PXE_IP_ADDR

This storage type is always big endian, not little endian.

```c
#pragma pack(1)
typedef union u_pxe_ip_addr {
    PXE_IPV6     IPv6;
    PXE_IPV4     IPv4;
} PXE_IP_ADDR;
#pragma pack()
```
E.3.5.6 PXE_DEVICE

This typedef is used to identify the network device that is being used by the UNDI. This information is returned by the Get Config Info command.

```c
#pragma pack(1)
typedef union pxe_device {

    // PCI and PC Card NICs are both identified using bus, device
    // and function numbers. For PC Card, this may require PC
    // Card services to be loaded in the BIOS or preboot
    // environment.
    struct {
        // See S/W UNDI ROMID structure definition for PCI and
        // PCC BusType definitions.
        PXE_UINT32  BusType;

        // Bus, device & function numbers that locate this device.
        PXE_UINT16  Bus;
        PXE_UINT8   Device;
        PXE_UINT8   Function;
    } PCI, PCC;

} PXE_DEVICE;
#pragma pack()
```

E.4 UNDI Commands

All 32/64-bit UNDI commands use the same basic command format, the CDB (Command Descriptor Block). CDB fields that are not used by a particular command must be initialized to zero by the application/driver that is issuing the command. (See “Links to UEFI-Related Documents” (http://uefi.org/uefi) under the heading “DMTF BIOS specifications”.)

All UNDI implementations must set the command completion status (**PXE_STATFLAGS_COMMAND_COMPLETE**) after command execution completes. Applications and drivers must not alter or rely on the contents of any of the CDB, CPB or DB fields until the command completion status is set.

All commands return status codes for invalid CDB contents and, if used, invalid CPB contents. Commands with invalid parameters will not execute. Fix the error and submit the command again.

**Figure E-6** describes the different UNDI states (Stopped, Started and Initialized), shows the transitions between the states and which UNDI commands are valid in each state.
Figure E-6 UNDI States, Transitions & Valid Commands

**Note:** All memory addresses including the CDB address, CPB address, and the DB address submitted to the S/W UNDI by the protocol drivers must be processor-based addresses. All memory addresses submitted to the H/W UNDI must be device based addresses.

**Note:** Additional requirements for S/W UNDI implementations: Processor register contents must be unchanged by S/W UNDI command execution (The application/driver does not have to save processor registers when calling S/W UNDI). Processor arithmetic flags are undefined (application/driver must save processor arithmetic flags if needed). Application/driver must remove CDB address from stack after control returns from S/W UNDI.

**Note:** Additional requirements for 32-bit network devices: All addresses given to the S/W UNDI must be 32-bit addresses. Any address that exceeds 32 bits (4 GiB) will result in a return of one of the following status codes: PXE_STATCODE_INVALID_PARAMETER, PXE_STATCODE_INVALID_CDB or PXE_STATCODE_INVALID_CPB.

When executing linked commands, command execution will stop at the end of the CDB list (when the PXE_CONTROL_LINK bit is not set) or when a command returns an error status code.
**Note:** Buffers requested via the MemoryRequired field in `s_pxe_db_get_init_info` (see Appendix E.4.5.5) will be allocated via `PCI_IO.AllocateBuffer()`. However, the buffers passed to various UNDI commands are not guaranteed to be allocated via `AllocateBuffer()`.

**Note:** Calls to `Map_Mem()` of type `TO_AND_FROM_DEVICE` must only be used for common DMA buffers. Such buffers must be requested via the MemoryRequired field in `s_pxe_db_get_init_info` and provided through the Initialize command.

### E.4.1 Command Linking and Queuing

When linking commands, the CDBs must be stored consecutively in system memory without any gaps in between. Do not set the Link bit in the last CDB in the list. As shown in Figure E-7, the Link bit must be set in all other CDBs in the list.

![Linked CDBs](image)

*Figure E-7 Linked CDBs*

When the H/W UNDI is executing commands, the State bits in the Status field in the IPXE structure will be set to Busy (3).

When H/W or S/W UNDI is executing commands and a new command is issued, a StatCode of `PXE_STATCODE_BUSY` and a StatFlag of `PXE_STATFLAG_COMMAND_FAILURE` is set in the CDB. For linked commands, only the first CDB will be set to Busy, all other CDBs will be unchanged. When a linked command fails, execution on the list stops. Commands after the failing command will not be run.

As shown in Figure E-8, when queuing commands, only the first CDB needs to have the Queue Control flag set. If queuing is supported and the UNDI is busy and there is room in the command queue, the command (or list of commands) will be queued.
When a command is queued a StatFlag of PXE_STATFLAG_COMMAND_QUEUED is set (if linked commands are queued only the StatFlag of the first CDB gets set). This signals that the command was added to the queue. Commands in the queue will be run on a first-in, first-out, basis. When a command fails, the next command in the queue is run. When a linked command in the queue fails, execution on the list stops. The next command, or list of commands, that was added to the command queue will be run.

**E.4.2 Get State**

This command is used to determine the operational state of the UNDI. An UNDI has three possible operational states:

- **Stopped.** A stopped UNDI is free for the taking. When all interface numbers (IFnum) for a particular S/W UNDI are stopped, that S/W UNDI image can be relocated or removed. A stopped UNDI will accept Get State and Start commands.

- **Started.** A started UNDI is in use. A started UNDI will accept Get State, Stop, Get Init Info, and Initialize commands.

- **Initialized.** An initialized UNDI is in use. An initialized UNDI will accept all commands except: Start, Stop, and Initialize.

Drivers and applications must not start using UNDIs that have been placed into the Started or Initialized states by another driver or application.

3.0 and 3.1 S/W UNDI: No callbacks are performed by this UNDI command.
### E.4.2.1 Issuing the Command

To issue a Get State command, create a CDB and fill it in as shown in the table below:

<table>
<thead>
<tr>
<th>CDB Field</th>
<th>How to initialize the CDB structure for a Get State command</th>
</tr>
</thead>
<tbody>
<tr>
<td>OpCode</td>
<td>PXE_OPCODE_GET_STATE</td>
</tr>
<tr>
<td>OpFlags</td>
<td>PXE_OPFLAGS_NOT_USED</td>
</tr>
<tr>
<td>CPBsize</td>
<td>PXE_CPBSIZE_NOT_USED</td>
</tr>
<tr>
<td>DBSize</td>
<td>PXE_DBSIZE_NOT_USED</td>
</tr>
<tr>
<td>CPBaddr</td>
<td>PXE_CPBADDR_NOT_USED</td>
</tr>
<tr>
<td>DBaddr</td>
<td>PXE_DBADDR_NOT_USED</td>
</tr>
<tr>
<td>StatCode</td>
<td>PXE_STATCODE_INITIALIZE</td>
</tr>
<tr>
<td>StatFlags</td>
<td>PXE_STATFLAGS_INITIALIZE</td>
</tr>
<tr>
<td>IFnum</td>
<td>A valid interface number from zero to ((!\text{PXE.IFcnt}</td>
</tr>
<tr>
<td>Control</td>
<td>Set as needed</td>
</tr>
</tbody>
</table>

### E.4.2.2 Waiting for the Command to Execute

Monitor the upper two bits (14 & 15) in the CDB.StatFlags field. Until these bits change to report PXE_STATFLAGS_COMMAND_COMPLETE or PXE_STATFLAGS_COMMAND_FAILED, the command has not been executed by the UNDI.

<table>
<thead>
<tr>
<th>StatFlags</th>
<th>Reason</th>
</tr>
</thead>
<tbody>
<tr>
<td>COMMAND_COMPLETE</td>
<td>Command completed successfully. StatFlags contain operational state.</td>
</tr>
<tr>
<td>COMMAND_QUEUED</td>
<td>Command has been queued. All other fields are unchanged.</td>
</tr>
<tr>
<td>INITIALIZE</td>
<td>Command has not been executed or queued.</td>
</tr>
</tbody>
</table>

### E.4.2.3 Checking Command Execution Results

After command execution completes, either successfully or not, the CDB.StatCode field contains the result of the command execution.

<table>
<thead>
<tr>
<th>StatCode</th>
<th>Reason</th>
</tr>
</thead>
<tbody>
<tr>
<td>SUCCESS</td>
<td>Command completed successfully. StatFlags contain operational state.</td>
</tr>
<tr>
<td>INVALID_CDB</td>
<td>One of the CDB fields was not set correctly.</td>
</tr>
<tr>
<td>BUSY</td>
<td>UNDI is already processing commands. Try again later.</td>
</tr>
<tr>
<td>QUEUE_FULL</td>
<td>Command queue is full. Try again later.</td>
</tr>
</tbody>
</table>
If the command completes successfully, use **PXE_STATFLAGS_GET_STATE_MASK** to check the state of the UNDI.

<table>
<thead>
<tr>
<th>StatFlags</th>
<th>Reason</th>
</tr>
</thead>
<tbody>
<tr>
<td>STOPPED</td>
<td>The UNDI is stopped.</td>
</tr>
<tr>
<td>STARTED</td>
<td>The UNDI is started, but not initialized.</td>
</tr>
<tr>
<td>INITIALIZED</td>
<td>The UNDI is initialized.</td>
</tr>
</tbody>
</table>

**E.4.3 Start**

This command is used to change the UNDI operational state from stopped to started. No other operational checks are made by this command. Protocol driver makes this call for each network interface supported by the UNDI with a set of call back routines and a unique identifier to identify the particular interface. UNDI does not interpret the unique identifier in any way except that it is a 64-bit value and it will pass it back to the protocol driver as a parameter to all the call back routines for any particular interface. If this is a S/W UNDI, the callback functions Delay(), Virt2Phys(), Map_Mem(), UnMap_Mem(), and Sync_Mem() functions will not be called by this command.

**E.4.3.1 Issuing the Command**

To issue a Start command for H/W UNDI, create a CDB and fill it in as shows in the table below:

<table>
<thead>
<tr>
<th>CDB Field</th>
<th>How to initialize the CDB structure for a H/W UNDI Start command</th>
</tr>
</thead>
<tbody>
<tr>
<td>OpCode</td>
<td>PXE_OPCODE_START</td>
</tr>
<tr>
<td>OpFlags</td>
<td>PXE_OPFLAGS_NOT_USED</td>
</tr>
<tr>
<td>CPBsize</td>
<td>PXE_CPBSIZE_NOT_USED</td>
</tr>
<tr>
<td>DBsize</td>
<td>PXE_DBSIZE_NOT_USED</td>
</tr>
<tr>
<td>CPBaddr</td>
<td>PXE_CPBADDR_NOT_USED</td>
</tr>
<tr>
<td>DBaddr</td>
<td>PXE_DBADDR_NOT_USED</td>
</tr>
<tr>
<td>StatCode</td>
<td>PXE_STATCODE_INITIALIZE</td>
</tr>
<tr>
<td>StatFlags</td>
<td>PXE_STATFLAGS_INITIALIZE</td>
</tr>
<tr>
<td>IFnum</td>
<td>A valid interface number from zero to (!PXE.IFcnt</td>
</tr>
<tr>
<td>Control</td>
<td>Set as needed</td>
</tr>
</tbody>
</table>
To issue a Start command for S/W UNDI, create a CDB and fill it in as shows in the table below:

<table>
<thead>
<tr>
<th>CDB Field</th>
<th>How to initialize the CDB structure for a S/W UNDI Start command</th>
</tr>
</thead>
<tbody>
<tr>
<td>OpCode</td>
<td>PXE_OPCODE_START</td>
</tr>
<tr>
<td>OpFlags</td>
<td>PXE_OPFLAGS_NOT_USED</td>
</tr>
<tr>
<td>CPBsize</td>
<td>sizeof(struct s_pxe_cpb_start_31)</td>
</tr>
<tr>
<td>DBsize</td>
<td>PXE_DBSIZE_NOT_USED</td>
</tr>
<tr>
<td>CPBaddr</td>
<td>Address of a PXE_CPB_START structure.</td>
</tr>
<tr>
<td>DBaddr</td>
<td>PXE_DBADDR_NOT_USED</td>
</tr>
<tr>
<td>StatCode</td>
<td>PXE_STATCODE_INITIALIZE</td>
</tr>
<tr>
<td>StatFlags</td>
<td>PXE_STATFLAGS_INITIALIZE</td>
</tr>
<tr>
<td>IFnum</td>
<td>A valid interface number from zero to (!PXE.IFcnt</td>
</tr>
<tr>
<td>Control</td>
<td>Set as needed</td>
</tr>
</tbody>
</table>

E.4.3.2 Preparing the CPB

For the 3.1 S/W UNDI Start command, the CPB structure shown below must be filled in and the CDB must be set to sizeof(struct s_pxe_cpb_start_31).

```c
#pragma pack(1)
typedef struct s_pxe_cpb_start_31 {
    UINT64  Delay;
    // // Address of the Delay() callback service.
    // // This field cannot be set to zero.
    // // // VOID
    // Delay(
    // IN UINT64  UniqueId,
    // IN UINT64  Microseconds);
    // // UNDI will never request a delay smaller than 10 microseconds
    // and will always request delays in increments of 10
    // microseconds. The Delay() callback routine must delay
    // between n and n + 10 microseconds before returning control
    // to the UNDI.
    // //
    
    UINT64  Block;
    // // Address of the Block() callback service.
    // // This field cannot be set to zero.
    // // // VOID
    // Block(
```
// IN UINT64 UniqueId,
// IN UINT32 Enable);
//
// UNDI may need to block multithreaded/multiprocessor access
// to critical code sections when programming or accessing the
// network device. When UNDI needs a block, it will call the
// Block() callback service with Enable set to a non-zero value.
// When UNDI no longer needs the block, it will call Block()
// with Enable set to zero.
//
UINT64 Virt2Phys;
//
// Convert a virtual address to a physical address.
// This field can be set to zero if virtual and physical
// addresses are identical.
//
// VOID
// Virt2Phys(
// IN UINT64 UniqueId,
// IN UINT64 Virtual,
// OUT UINT64 PhysicalPtr);
//
// UNDI will pass in a virtual address and a pointer to storage
// for a physical address. The Virt2Phys() service converts
// the virtual address to a physical address and stores the
// resulting physical address in the supplied buffer. If no
// conversion is needed, the virtual address must be copied
// into the supplied physical address buffer.
//
UINT64 Mem_IO;
//
// Read/Write network device memory and/or I/O register space.
// This field cannot be set to zero.
//
// VOID
// Mem_IO(
// IN UINT64 UniqueId,
// IN UINT8 AccessType,
// IN UINT8 Length,
// IN UINT64 Port,
// IN OUT UINT64 BufferPtr);
//
// UNDI uses the Mem_IO() service to access the network device
// memory and/or I/O registers. The AccessType is one of the
// PXE_IO_xxx or PXE_MEM_xxx constants defined at the end of
// this section. The Length is 1, 2, 4 or 8. The Port number
// is relative to the base memory or I/O address space for this
// device. BufferPtr points to the data to be written to the
// Port or will contain the data that is read from the Port.
//
UINT64 Map_Mem;
//
// Map virtual memory address for DMA.
// This field can be set to zero if there is no mapping
// service.
//
// VOID
// Map_Mem(
//  IN UINT64 UniqueId,
//  IN UINT64 Virtual,
//  IN UINT32 Size,
//  IN UINT32 Direction,
//  OUT UINT64 PhysicalPtr);
//
// When UNDI needs to perform a DMA transfer it will request a
// virtual-to-physical mapping using the Map_Mem() service. The
// Virtual parameter contains the virtual address to be mapped.
// The minimum Size of the virtual memory buffer to be mapped.
// Direction is one of the TO_DEVICE, FROM_DEVICE or
// TO_AND_FROM_DEVICE constants defined at the end of this
// section. PhysicalPtr contains the mapped physical address or
// a copy of the Virtual address if no mapping is required.
//
UINT64 UnMap_Mem;
//
// Un-map previously mapped virtual memory address.
// This field can be set to zero only if the Map_Mem() service
// is also set to zero.
//
// VOID
// UnMap_Mem(
//  IN UINT64 UniqueId,
//  IN UINT64 Virtual,
//  IN UINT32 Size,
//  IN UINT32 Direction,
//  IN UINT64 PhysicalPtr);
//
// When UNDI is done with the mapped memory, it will use the
// UnMap_Mem() service to release the mapped memory.
UINT64 Sync_Mem;
//
// Synchronise mapped memory.
// This field can be set to zero only if the Map_Mem() service
// is also set to zero.
//
// VOID
// Sync_Mem(
//  IN UINT64  UniqueId,
//  IN UINT64  Virtual,
//  IN UINT32  Size,
//  IN UINT32  Direction,
//  IN UINT64  PhysicalPtr);
//
// When the virtual and physical buffers need to be
// synchronized, UNDI will call the Sync_Mem() service.
//
UINT64 UniqueId;
//
// UNDI will pass this value to each of the callback services.
// A unique ID number should be generated for each instance of
// the UNDI driver that will be using these callback services.
//
} PXE_CPB_START_31;
#pragma pack()
#pragma pack(1)

typedef struct s_pxe_cpb_start_30 {
  UINT64  Delay;
  //
  // Address of the Delay() callback service.
  // This field cannot be set to zero.
  //
  // VOID
  // Delay(
  //  IN UINT64  Microseconds);
  //
  // UNDI will never request a delay smaller than 10 microseconds
  // and will always request delays in increments of 10.
  // microseconds The Delay() callback routine must delay between
  // n and n + 10 microseconds before returning control to the
  // UNDI.
  //
  UINT64  Block;
  //
  // Address of the Block() callback service.
  // This field cannot be set to zero.
  //
  // VOID
  // Block(
  //  IN UINT32  Enable);
  //
  // UNDI may need to block multithreaded/multiprocessor access
  // to critical code sections when programming or accessing the
  // network device. When UNDI needs a block, it will call the
  // Block()callback service with Enable set to a non-zero value.
  // When UNDI no longer needs the block, it will call Block()
  // with Enable set to zero.
  //
  UINT64  Virt2Phys;
  //
  // Convert a virtual address to a physical address.
  // This field can be set to zero if virtual and physical
  // addresses are identical.
  //
  // VOID
  // Virt2Phys(
  //  IN UINT64  Virtual,
  //  OUT UINT64  PhysicalPtr);
  //
  // UNDI will pass in a virtual address and a pointer to storage
// for a physical address. The Virt2Phys() service converts
// the virtual address to a physical address and stores the
// resulting physical address in the supplied buffer. If no
// conversion is needed, the virtual address must be copied
// into the supplied physical address buffer.

UINT64 Mem_IO;

// Read/Write network device memory and/or I/O register space.
// This field cannot be set to zero.

VOID Mem_IO(
    IN UINT8AccessType,
    IN UINT8Length,
    IN UINT64Port,
    IN OUT UINT64BufferPtr);

UNDI uses the Mem_IO() service to access the network device
memory and/or I/O registers. The AccessType is one of the
PXE_IO_xxx or PXE_MEM_xxx constants defined at the end of
this section. The Length is 1, 2, 4 or 8. The Port number
is relative to the base memory or I/O address space for this
device. BufferPtr points to the data to be written to the
Port or will contain the data that is read from the Port.

#pragma pack() gunman
#define TO_AND_FROM_DEVICE 0
// Provides both read and write access to system memory by both
// the processor and a bus master. The buffer is coherent from
// both the processor's and the bus master's point of view.

#define FROM_DEVICE 1
// Provides a write operation to system memory by a bus master.

#define TO_DEVICE 2
// Provides a read operation from system memory by a bus master.
E.4.3.3 Waiting for the Command to Execute

Monitor the upper two bits (14 & 15) in the CDB.StatFlags field. Until these bits change to report PXE_STATFLAGS_COMMAND_COMPLETE or PXE_STATFLAGS_COMMAND_FAILED, the command has not been executed by the UNDI.

<table>
<thead>
<tr>
<th>StatFlags</th>
<th>Reason</th>
</tr>
</thead>
<tbody>
<tr>
<td>COMMAND_COMPLETE</td>
<td>Command completed successfully. UNDI is now started.</td>
</tr>
<tr>
<td>COMMAND_QUEUED</td>
<td>Command has been queued.</td>
</tr>
<tr>
<td>INITIALIZE</td>
<td>Command has been not executed or queued.</td>
</tr>
</tbody>
</table>

E.4.3.4 Checking Command Execution Results

After command execution completes, either successfully or not, the CDB.StatCode field contains the result of the command execution.

<table>
<thead>
<tr>
<th>StatCode</th>
<th>Reason</th>
</tr>
</thead>
<tbody>
<tr>
<td>SUCCESS</td>
<td>Command completed successfully. UNDI is now started.</td>
</tr>
<tr>
<td>INVALID_CDB</td>
<td>One of the CDB fields was not set correctly.</td>
</tr>
<tr>
<td>BUSY</td>
<td>UNDI is already processing commands. Try again later.</td>
</tr>
<tr>
<td>QUEUE_FULL</td>
<td>Command queue is full. Try again later.</td>
</tr>
<tr>
<td>ALREADY_STARTED</td>
<td>The UNDI is already started.</td>
</tr>
</tbody>
</table>

E.4.4 Stop

This command is used to change the UNDI operational state from started to stopped.

E.4.4.1 Issuing the Command

To issue a Stop command, create a CDB and fill it in as shows in the table below:

<table>
<thead>
<tr>
<th>CDB Field</th>
<th>How to initialize the CDB structure for a Stop command</th>
</tr>
</thead>
<tbody>
<tr>
<td>OpCode</td>
<td>PXE_OPCODE_STOP</td>
</tr>
<tr>
<td>OpFlags</td>
<td>PXE_OPFLAGS_NOT_USED</td>
</tr>
<tr>
<td>CPBsize</td>
<td>PXE_CPSIZE_NOT_USED</td>
</tr>
<tr>
<td>DBsize</td>
<td>PXE_DBSIZE_NOT_USED</td>
</tr>
<tr>
<td>CPBaddr</td>
<td>PXE_CPBADDR_NOT_USED</td>
</tr>
<tr>
<td>DBaddr</td>
<td>PXE_DBADDR_NOT_USED</td>
</tr>
<tr>
<td>StatCode</td>
<td>PXE_STATCODE_INITIALIZE</td>
</tr>
<tr>
<td>StatFlags</td>
<td>PXE_STATFLAGS_INITIALIZE</td>
</tr>
<tr>
<td>IFnum</td>
<td>A valid interface number from zero to (!PXE.IFCnt</td>
</tr>
<tr>
<td>Control</td>
<td>Set as needed</td>
</tr>
</tbody>
</table>
E.4.4.2 Waiting for the Command to Execute

Monitor the upper two bits (14 & 15) in the \texttt{CDB.StatFlags} field. Until these bits change to report \texttt{PXE_STATFLAGS_COMMAND_COMPLETE} or \texttt{PXE_STATFLAGS_COMMAND_FAILED}, the command has not been executed by the UNDI.

<table>
<thead>
<tr>
<th>StatFlags</th>
<th>Reason</th>
</tr>
</thead>
<tbody>
<tr>
<td>COMMAND_COMPLETE</td>
<td>Command completed successfully. UNDI is now stopped.</td>
</tr>
<tr>
<td>COMMAND_QUEUED</td>
<td>Command has been queued.</td>
</tr>
<tr>
<td>INITIALIZE</td>
<td>Command has not been executed or queued.</td>
</tr>
</tbody>
</table>

E.4.4.3 Checking Command Execution Results

After command execution completes, either successfully or not, the \texttt{CDB.StatCode} field contains the result of the command execution.

<table>
<thead>
<tr>
<th>StatCode</th>
<th>Reason</th>
</tr>
</thead>
<tbody>
<tr>
<td>SUCCESS</td>
<td>Command completed successfully. UNDI is now stopped.</td>
</tr>
<tr>
<td>INVALID_CDB</td>
<td>One of the CDB fields was not set correctly.</td>
</tr>
<tr>
<td>BUSY</td>
<td>UNDI is already processing commands. Try again later.</td>
</tr>
<tr>
<td>QUEUE_FULL</td>
<td>Command queue is full. Try again later.</td>
</tr>
<tr>
<td>NOT_STARTED</td>
<td>The UNDI is not started.</td>
</tr>
<tr>
<td>NOT_SHUTDOWN</td>
<td>The UNDI is initialized and must be shutdown before it can be stopped.</td>
</tr>
</tbody>
</table>

E.4.5 Get Init Info

This command is used to retrieve initialization information that is needed by drivers and applications to initialized UNDI.

E.4.5.1 Issuing the Command

To issue a Get Init Info command, create a CDB and fill it in as shows in the table below:

<table>
<thead>
<tr>
<th>CDB Field</th>
<th>How to initialize the CDB structure for a Get Init Info command</th>
</tr>
</thead>
<tbody>
<tr>
<td>OpCode</td>
<td>\texttt{PXE_OPCODE_GET_INIT_INFO}</td>
</tr>
<tr>
<td>OpFlags</td>
<td>\texttt{PXE_OPFLAGS_NOT_USED}</td>
</tr>
<tr>
<td>CPBsize</td>
<td>\texttt{PXE_CPBSIZE_NOT_USED}</td>
</tr>
<tr>
<td>DBsize</td>
<td>sizeof(\texttt{PXE_DB_INIT_INFO})</td>
</tr>
<tr>
<td>CPBaddr</td>
<td>\texttt{PXE_CPBADDR_NOT_USED}</td>
</tr>
<tr>
<td>DBaddr</td>
<td>Address of a \texttt{PXE_DB_INIT_INFO} structure.</td>
</tr>
<tr>
<td>StatCode</td>
<td>\texttt{PXE_STATCODE_INITIALIZE}</td>
</tr>
<tr>
<td>StatFlags</td>
<td>\texttt{PXE_STATFLAGS_INITIALIZE}</td>
</tr>
<tr>
<td>IFnum</td>
<td>A valid interface number from zero to ((!\texttt{PXE.IFcnt}</td>
</tr>
</tbody>
</table>
E.4.5.2 Waiting for the Command to Execute

Monitor the upper two bits (14 & 15) in the CDB.StatFlags field. Until these bits change to report PXE_STATFLAGS_COMMAND_COMPLETE or PXE_STATFLAGS_COMMAND_FAILED, the command has not been executed by the UNDI.

<table>
<thead>
<tr>
<th>StatFlags</th>
<th>Reason</th>
</tr>
</thead>
<tbody>
<tr>
<td>COMMAND_COMPLETE</td>
<td>Command completed successfully. DB can be used.</td>
</tr>
<tr>
<td>COMMAND_QUEUED</td>
<td>Command has been queued.</td>
</tr>
<tr>
<td>INITIALIZE</td>
<td>Command has been not executed or queued.</td>
</tr>
</tbody>
</table>

E.4.5.3 Checking Command Execution Results

After command execution completes, either successfully or not, the CDB.StatCode field contains the result of the command execution.

<table>
<thead>
<tr>
<th>StatCode</th>
<th>Reason</th>
</tr>
</thead>
<tbody>
<tr>
<td>SUCCESS</td>
<td>Command completed successfully. DB can be used.</td>
</tr>
<tr>
<td>INVALID_CDB</td>
<td>One of the CDB fields was not set correctly.</td>
</tr>
<tr>
<td>BUSY</td>
<td>UNDI is already processing commands. Try again later.</td>
</tr>
<tr>
<td>QUEUE_FULL</td>
<td>Command queue is full. Try again later.</td>
</tr>
<tr>
<td>NOT_STARTED</td>
<td>The UNDI is not started.</td>
</tr>
</tbody>
</table>

E.4.5.4 StatFlags

To determine if cable detection is supported by this UNDI/NIC, use these macros with the value returned in the CDB.StatFlags field:

```
PXE_STATFLAGS_CABLE_DETECT_MASK
PXE_STATFLAGS_CABLE_DETECT_NOT_SUPPORTED
PXE_STATFLAGS_CABLE_DETECT_SUPPORTED
PXE_STATFLAGS_GET_STATUS_NO_MEDIA_MASK
PXE_STATFLAGS_GET_STATUS_NO_MEDIA_NOT_SUPPORTED
PXE_STATFLAGS_GET_STATUS_NO_MEDIA_SUPPORTED
```

E.4.5.5 DB

```
#pragma pack(1)
typedef struct s_pxe_db_get_init_info {

    // Minimum length of locked memory buffer that must be given to
    // the Initialize command. Giving UNDI more memory will
    // generally give better performance.

    // If MemoryRequired is zero, the UNDI does not need and will
```
// not use system memory to receive and transmit packets.

PXE_UINT32 MemoryRequired;

// Maximum frame data length for Tx/Rx excluding the media
// header.
//
PXE_UINT32 FrameDataLen;

// Supported link speeds are in units of mega bits. Common
// ethernet values are 10, 100 and 1000. Unused LinkSpeeds[]
// entries are zero filled.

PXE_UINT32 LinkSpeeds[4];

// Number of nonvolatile storage items.

PXE_UINT32 NvCount;

// Width of nonvolatile storage item in bytes. 0, 1, 2 or 4

PXE_UINT16 NvWidth;

// Media header length. This is the typical media header
// length for this UNDI. This information is needed when
// allocating receive and transmit buffers.

PXE_UINT16 MediaHeaderLen;

// Number of bytes in the NIC hardware (MAC) address.

PXE_UINT16 HWaddrLen;

// Maximum number of multicast MAC addresses in the multicast
// MAC address filter list.

PXE_UINT16 MCastFilterCnt;

// Default number and size of transmit and receive buffers that
// will be allocated by the UNDI. If MemoryRequired is
// nonzero, this allocation will come out of the memory buffer
// given to the Initialize command. If MemoryRequired is zero,
// this allocation will come out of memory on the NIC.

PXE_UINT16 TxBufCnt;
PXE_UINT16 TxBufSize;
PXE_UINT16 RxBufCnt;
PXE_UINT16 RxBufSize;
// Hardware interface types defined in the Assigned Numbers RFC
// and used in DHCP and ARP packets.
// See the PXE_IFTYPE typedef and PXE_IFTYPE_xxx macros.

PXE_UINT8 IFtype;

// Supported duplex options. This can be one or a combination
// of more than one constants defined as PXE_DUPLEX_xxxxx
// below. This value indicates the ability of UNDI to
// change/control the duplex modes of the NIC.

PXE_UINT8 SupportedDuplexModes;

// Supported loopback options. This field can be one or a
// combination of more than one constants defined as
// PXE_LOOPBACK_xxxxx #defines below. This value indicates
// the ability of UNDI to change/control the loopback modes
// of the NIC

PXE_UINT8 SupportedLoopBackModes;

} PXE_DB_GET_INIT_INFO;
#pragma pack()
E.4.6.2 Waiting for the Command to Execute

Monitor the upper two bits (14 & 15) in the `CDB.StatFlags` field. Until these bits change to report `PXE_STATFLAGS_COMMAND_COMPLETE` or `PXE_STATFLAGS_COMMAND_FAILED`, the command has not been executed by the UNDI.

<table>
<thead>
<tr>
<th>StatFlags</th>
<th>Reason</th>
</tr>
</thead>
<tbody>
<tr>
<td>COMMAND_COMPLETE</td>
<td>Command completed successfully. DB has been written.</td>
</tr>
<tr>
<td>COMMAND_QUEUED</td>
<td>Command has been queued.</td>
</tr>
<tr>
<td>INITIALIZE</td>
<td>Command has been not executed or queued.</td>
</tr>
</tbody>
</table>

E.4.6.3 Checking Command Execution Results

After command execution completes, either successfully or not, the `CDB.StatCode` field contains the result of the command execution.

<table>
<thead>
<tr>
<th>StatCode</th>
<th>Reason</th>
</tr>
</thead>
<tbody>
<tr>
<td>SUCCESS</td>
<td>Command completed successfully. DB has been written.</td>
</tr>
<tr>
<td>INVALID_CDB</td>
<td>One of the CDB fields was not set correctly.</td>
</tr>
<tr>
<td>BUSY</td>
<td>UNDI is already processing commands. Try again later.</td>
</tr>
<tr>
<td>QUEUE_FULL</td>
<td>Command queue is full. Try again later.</td>
</tr>
<tr>
<td>NOT_STARTED</td>
<td>The UNDI is not started.</td>
</tr>
</tbody>
</table>

E.4.6.4 DB

```c
#pragma pack(1)
typedef struct s_pxe_pci_config_info {

    // This is the flag field for the PXE_DB_GET_CONFIG_INFO union.
    // For PCI bus devices, this field is set to PXE_BUSTYPE_PCI.

    PXE_UINT32 BusType;

    // This identifies the PCI network device that this UNDI
    // interface is bound to.
```
PXE_UINT16 Bus;
PXE_UINT8 Device;
PXE_UINT8 Function;

// This is a copy of the PCI configuration space for this
// network device.
union {
    PXE_UINT8 Byte[256];
    PXE_UINT16 Word[128];
    PXE_UINT32 Dword[64];
} Config;
} PXE_PCI_CONFIG_INFO;
#pragma pack()
#pragma pack(1)
typedef struct s_pxe_pcc_config_info {
    // This is the flag field for the PXE_DB_GET_CONFIG_INFO union.
    // For PCC bus devices, this field is set to PXE_BUSTYPE_PCC.
    PXE_UINT32 BusType;

    // This identifies the PCC network device that this UNDI
    // interface is bound to.
    PXE_UINT16 Bus;
PXE_UINT8 Device;
PXE_UINT8 Function;

    // This is a copy of the PCC configuration space for this
    // network device.
    union {
        PXE_UINT8 Byte[256];
        PXE_UINT16 Word[128];
        PXE_UINT32 Dword[64];
    } Config;
} PXE_PCC_CONFIG_INFO;
#pragma pack()
#pragma pack(1)
typedef union u_pxe_db_get_config_info {
    PXE_PCI_CONFIG_INFO pci;
PXE_PCC_CONFIG_INFO pcc;
} PXE_DB_GET_CONFIG_INFO;
#pragma pack()
E.4.7 Initialize

This command resets the network adapter and initializes UNDI using the parameters supplied in the CPB. The Initialize command must be issued before the network adapter can be setup to transmit and receive packets. This command will not enable the receive unit or external interrupts.

Once the memory requirements of the UNDI are obtained by using the Get Init Info command, a block of kernel (nonswappable) memory may need to be allocated by the protocol driver. The address of this kernel memory must be passed to UNDI using the Initialize command CPB. This memory is used for transmit and receive buffers and internal processing.

Initializing the network device will take up to four seconds for most network devices and in some extreme cases (usually poor cables) up to twenty seconds. Control will not be returned to the caller and the `COMMAND_COMPLETE` status flag will not be set until the NIC is ready to transmit.

E.4.7.1 Issuing the Command

To issue an Initialize command, create a CDB and fill it in as shows in the table below:

<table>
<thead>
<tr>
<th>CDB Field</th>
<th>How to initialize the CDB structure for an Initialize command</th>
</tr>
</thead>
<tbody>
<tr>
<td>OpCode</td>
<td>PXE_OPCODE_INITIALIZE</td>
</tr>
<tr>
<td>OpFlags</td>
<td>Set as needed.</td>
</tr>
<tr>
<td>CPBsize</td>
<td>sizeof(PXE_CPB_INITIALIZE)</td>
</tr>
<tr>
<td>DBsize</td>
<td>sizeof(PXE_DB_INITIALIZE)</td>
</tr>
<tr>
<td>CPBaddr</td>
<td>Address of a PXE_CPB_INITIALIZE structure.</td>
</tr>
<tr>
<td>Dbaddr</td>
<td>Address of a PXE_DB_INITIALIZE structure.</td>
</tr>
<tr>
<td>StatCode</td>
<td>PXE_STATCODE_INITIALIZE</td>
</tr>
<tr>
<td>StatFlags</td>
<td>PXE_STATFLAGS_INITIALIZE</td>
</tr>
<tr>
<td>Ifnum</td>
<td>A valid interface number from zero to (!PXE.IFcnt</td>
</tr>
<tr>
<td>Control</td>
<td>Set as needed.</td>
</tr>
</tbody>
</table>

E.4.7.2 OpFlags

Cable detection can be enabled or disabled by setting one of the following OpFlags:

- PXE_OPFLAGS_INITIALIZE_CABLE_DETECT
- PXE_OPFLAGS_INITIALIZE_DO_NOT_DETECT_CABLE

E.4.7.3 Preparing the CPB

If the `MemoryRequired` field returned in the PXE_DB_GET_INIT_INFO structure is zero, the Initialize command does not need to be given a memory buffer or even a CPB structure. If the `MemoryRequired` field is nonzero, the Initialize command does need a memory buffer.

```c
#pragma pack(1)
typedef struct s_pxe_cpb_initialize {
```
// Address of first (lowest) byte of the memory buffer.
// This buffer must be in contiguous physical memory and cannot
// be swapped out. The UNDI will be using this for transmit
// and receive buffering. This address must be a processor-
// based address for S/W UNDI and a device-based address for
// H/W UNDI.

PXE_UINT64 MemoryAddr;

// MemoryLength must be greater than or equal to MemoryRequired
// returned by the Get Init Info command.

PXE_UINT32 MemoryLength;

// Desired link speed in Mbit/sec. Common ethernet values are
// 10, 100 and 1000. Setting a value of zero will auto-detect
// and/or use the default link speed (operation depends on
// UNDI/NIC functionality).

PXE_UINT32 LinkSpeed;

// Suggested number and size of receive and transmit buffers to
// allocate. If MemoryAddr and MemoryLength are nonzero, this
// allocation comes out of the supplied memory buffer. If
// MemoryAddr and MemoryLength are zero, this allocation comes
// out of memory on the NIC.

// If these fields are set to zero, the UNDI will allocate
// buffer counts and sizes as it sees fit.

PXE_UINT16 TxBufCnt;
PXE_UINT16 TxBufSize;
PXE_UINT16 RxBufCnt;
PXE_UINT16 RxBufSize;

// The following configuration parameters are optional and must
// be zero to use the default values.
// The possible values for these parameters are defined below.

PXE_UINT8 DuplexMode;
PXE_UINT8 LoopBackMode;

// define PXE_DUPLEX_AUTO_DETECT 0x00
#define PXE_FORCE_FULL_DUPLEX 0x01
#define PXE_FORCE_HALF_DUPLEX 0x02


```c
#define PXE_LOOPBACK_NORMAL 0
#define PXE_LOOPBACK_INTERNAL 1
#define PXE_LOOPBACK_EXTERNAL 2
```

### E.4.7.4 Waiting for the Command to Execute

Monitor the upper two bits (14 & 15) in the `CDB.StatFlags` field. Until these bits change to report `PXE_STATFLAGS_COMMAND_COMPLETE` or `PXE_STATFLAGS_COMMAND_FAILED`, the command has not been executed by the UNDI.

<table>
<thead>
<tr>
<th>StatFlags</th>
<th>Reason</th>
</tr>
</thead>
<tbody>
<tr>
<td>COMMAND_COMPLETE</td>
<td>Command completed successfully. UNDI and network device is now initialized. DB has been written.</td>
</tr>
<tr>
<td>COMMAND_QUEUED</td>
<td>Command has been queued.</td>
</tr>
<tr>
<td>INITIALIZE</td>
<td>Command has been not executed or queued.</td>
</tr>
</tbody>
</table>

### E.4.7.5 Checking Command Execution Results

After command execution completes, either successfully or not, the `CDB.StatCode` field contains the result of the command execution.

<table>
<thead>
<tr>
<th>StatCode</th>
<th>Reason</th>
</tr>
</thead>
<tbody>
<tr>
<td>SUCCESS</td>
<td>Command completed successfully. UNDI and network device is now initialized. DB has been written. Check StatFlags.</td>
</tr>
<tr>
<td>INVALID_CDB</td>
<td>One of the CDB fields was not set correctly.</td>
</tr>
<tr>
<td>INVALID_CPB</td>
<td>One of the CPB fields was not set correctly.</td>
</tr>
<tr>
<td>BUSY</td>
<td>UNDI is already processing commands. Try again later.</td>
</tr>
<tr>
<td>QUEUE_FULL</td>
<td>Command queue is full. Try again later.</td>
</tr>
<tr>
<td>NOT_STARTED</td>
<td>The UNDI is not started.</td>
</tr>
<tr>
<td>ALREADY_INITIALIZED</td>
<td>The UNDI is already initialized.</td>
</tr>
<tr>
<td>DEVICE_FAILURE</td>
<td>The network device could not be initialized.</td>
</tr>
<tr>
<td>NVDATA_FAILURE</td>
<td>The nonvolatile storage could not be read.</td>
</tr>
</tbody>
</table>

### E.4.7.6 StatFlags

Check the StatFlags to see if there is an active connection to this network device. If the no media StatFlag is set, the UNDI and network device are still initialized.
PXE_STATFLAGS_INITIALIZED_NO_MEDIA

E.4.7.7 Before Using the DB

#define pack(1)
typedef struct s_pxe_db_initialize {
  // Actual amount of memory used from the supplied memory
  // buffer. This may be less that the amount of memory
  // supplied and may be zero if the UNDI and network device
  // do not use external memory buffers. Memory used by the
  // UNDI and network device is allocated from the lowest
  // memory buffer address.

  PXE_UINT32 MemoryUsed;

  // Actual number and size of receive and transmit buffers that
  // were allocated.

  PXE_UINT16 TxBufCnt;
  PXE_UINT16 TxBufSize;
  PXE_UINT16 RxBufCnt;
  PXE_UINT16 RxBufSize
} PXE_DB_INITIALIZE;
#define pack()
E.4.8.2 OpFlags

Normally the settings of the receive filters and external interrupt enables are unchanged by the Reset command. These two OpFlags will alter the operation of the Reset command.

- PXE_OPFLAGS_RESET_DISABLE_INTERRUPTS
- PXE_OPFLAGS_RESET_DISABLE_FILTERS

E.4.8.3 Waiting for the Command to Execute

Monitor the upper two bits (14 & 15) in the CDB.StatFlags field. Until these bits change to report PXE_STATFLAGS_COMMAND_COMPLETE or PXE_STATFLAGS_COMMAND_FAILED, the command has not been executed by the UNDI.

<table>
<thead>
<tr>
<th>StatFlags</th>
<th>Reason</th>
</tr>
</thead>
<tbody>
<tr>
<td>COMMAND_COMPLETE</td>
<td>Command completed successfully. UNDI and network device have been reset. Check StatFlags.</td>
</tr>
<tr>
<td>COMMAND_QUEUED</td>
<td>Command has been queued.</td>
</tr>
<tr>
<td>INITIALIZE</td>
<td>Command has been not executed or queued.</td>
</tr>
</tbody>
</table>

E.4.8.4 Checking Command Execution Results

After command execution completes, either successfully or not, the CDB.StatCode field contains the result of the command execution.

<table>
<thead>
<tr>
<th>StatCode</th>
<th>Reason</th>
</tr>
</thead>
<tbody>
<tr>
<td>SUCCESS</td>
<td>Command completed successfully. UNDI and network device have been reset. Check StatFlags.</td>
</tr>
<tr>
<td>INVALID_CDB</td>
<td>One of the CDB fields was not set correctly.</td>
</tr>
<tr>
<td>BUSY</td>
<td>UNDI is already processing commands. Try again later.</td>
</tr>
<tr>
<td>QUEUE_FULL</td>
<td>Command queue is full. Try again later.</td>
</tr>
<tr>
<td>NOT_STARTED</td>
<td>The UNDI is not started.</td>
</tr>
<tr>
<td>NOT_INITIALIZED</td>
<td>The UNDI is not initialized.</td>
</tr>
<tr>
<td>DEVICE_FAILURE</td>
<td>The network device could not be initialized.</td>
</tr>
<tr>
<td>NVDATA_FAILURE</td>
<td>The nonvolatile storage is not valid.</td>
</tr>
</tbody>
</table>

E.4.8.5 StatFlags

Check the StatFlags to see if there is an active connection to this network device. If the no media StatFlag is set, the UNDI and network device are still reset.
E.4.9 Shutdown

The Shutdown command resets the network adapter and leaves it in a safe state for another driver to initialize. Any pending transmits or receives are lost. Receive filters and external interrupt enables are reset (disabled). The memory buffer assigned in the Initialize command can be released or reassigned.

Once UNDI has been shutdown, it can then be stopped or initialized again. The Shutdown command changes the UNDI operational state from initialized to started.

E.4.9.1 Issuing the Command

To issue a Shutdown command, create a CDB and fill it in as shown in the table below:

<table>
<thead>
<tr>
<th>CDB Field</th>
<th>How to initialize the CDB structure for a Shutdown command</th>
</tr>
</thead>
<tbody>
<tr>
<td>OpCode</td>
<td>PXE_OPCODE_SHUTDOWN</td>
</tr>
<tr>
<td>OpFlags</td>
<td>PXE_OPFLAGS_NOT_USED</td>
</tr>
<tr>
<td>CPBsize</td>
<td>PXE_CPBSIZE_NOT_USED</td>
</tr>
<tr>
<td>DBsize</td>
<td>PXE_DBSIZE_NOT_USED</td>
</tr>
<tr>
<td>CPBaddr</td>
<td>PXE_CPBSIZE_NOT_USED</td>
</tr>
<tr>
<td>DBaddr</td>
<td>PXE_DBSIZE_NOT_USED</td>
</tr>
<tr>
<td>StatCode</td>
<td>PXE_STATCODE_INITIALIZE</td>
</tr>
<tr>
<td>StatFlags</td>
<td>PXE_STATFLAGS_INITIALIZE</td>
</tr>
<tr>
<td>IFnum</td>
<td>A valid interface number from zero to ('!PXE.IFcnt</td>
</tr>
<tr>
<td>Control</td>
<td>Set as needed.</td>
</tr>
</tbody>
</table>

E.4.9.2 Waiting for the Command to Execute

Monitor the upper two bits (14 & 15) in the CDB.StatFlags field. Until these bits change to report PXE_STATFLAGS_COMMAND_COMPLETE or PXE_STATFLAGS_COMMAND_FAILED, the command has not been executed by the UNDI.

<table>
<thead>
<tr>
<th>StatFlags</th>
<th>Reason</th>
</tr>
</thead>
<tbody>
<tr>
<td>COMMAND_COMPLETE</td>
<td>Command completed successfully. UNDI and network device are shutdown.</td>
</tr>
<tr>
<td>COMMAND_QUEUE</td>
<td>Command has been queued.</td>
</tr>
<tr>
<td>INITIALIZE</td>
<td>Command has been not executed or queued.</td>
</tr>
</tbody>
</table>
E.4.9.3 Checking Command Execution Results

After command execution completes, either successfully or not, the `CDB.StatCode` field contains the result of the command execution.

<table>
<thead>
<tr>
<th>StatCode</th>
<th>Reason</th>
</tr>
</thead>
<tbody>
<tr>
<td>SUCCESS</td>
<td>Command completed successfully. UNDI and network device are shutdown.</td>
</tr>
<tr>
<td>INVALID_CDB</td>
<td>One of the CDB fields was not set correctly.</td>
</tr>
<tr>
<td>BUSY</td>
<td>UNDI is already processing commands. Try again later.</td>
</tr>
<tr>
<td>QUEUE_FULL</td>
<td>Command queue is full. Try again later.</td>
</tr>
<tr>
<td>NOT_STARTED</td>
<td>The UNDI is not started.</td>
</tr>
<tr>
<td>NOT_INITIALIZED</td>
<td>The UNDI is not initialized.</td>
</tr>
</tbody>
</table>

E.4.10 Interrupt Enables

The Interrupt Enables command can be used to read and/or change the current external interrupt enable settings. Disabling an external interrupt enable prevents an external (hardware) interrupt from being signaled by the network device, internally the interrupt events can still be polled by using the Get Status command.

E.4.10.1 Issuing the Command

To issue an Interrupt Enables command, create a CDB and fill it in as shows in the table below:

<table>
<thead>
<tr>
<th>CDB Field</th>
<th>How to initialize the CDB structure for an Interrupt Enables command</th>
</tr>
</thead>
<tbody>
<tr>
<td>OpCode</td>
<td>PXE_OPCODE_INTERRUPT_ENABLES</td>
</tr>
<tr>
<td>OpFlags</td>
<td>Set as needed.</td>
</tr>
<tr>
<td>CPBsize</td>
<td>PXE_CPBSIZE_NOT_USED</td>
</tr>
<tr>
<td>DBsize</td>
<td>PXE_DBSIZE_NOT_USED</td>
</tr>
<tr>
<td>CPBaddr</td>
<td>PXE_CPBADDR_NOT_USED</td>
</tr>
<tr>
<td>DBaddr</td>
<td>PXE_DBADDR_NOT_USED</td>
</tr>
<tr>
<td>StatCode</td>
<td>PXE_STATCODE_INITIALIZE</td>
</tr>
<tr>
<td>StatFlags</td>
<td>PXE_STATFLAGS_INITIALIZE</td>
</tr>
<tr>
<td>IFnum</td>
<td>A valid interface number from zero to `(!PXE.IFcnt</td>
</tr>
<tr>
<td>Control</td>
<td>Set as needed.</td>
</tr>
</tbody>
</table>
E.4.10.2 OpFlags

To read the current external interrupt enables settings set \texttt{CDB.OpFlags} to:

\texttt{PXE\_OPFLAGS\_INTERRUPT\_READ}

To enable or disable external interrupts set one of these OpFlags:

\texttt{PXE\_OPFLAGS\_INTERRUPT\_DISABLE}
\texttt{PXE\_OPFLAGS\_INTERRUPT\_ENABLE}

When enabling or disabling interrupt settings, the following additional OpFlag bits are used to specify which types of external interrupts are to be enabled or disabled:

\texttt{PXE\_OPFLAGS\_INTERRUPT\_RECEIVE}
\texttt{PXE\_OPFLAGS\_INTERRUPT\_TRANSMIT}
\texttt{PXE\_OPFLAGS\_INTERRUPT\_COMMAND}
\texttt{PXE\_OPFLAGS\_INTERRUPT\_SOFTWARE}

Setting \texttt{PXE\_OPFLAGS\_INTERRUPT\_SOFTWARE} does not enable an external interrupt type, it generates an external interrupt.

E.4.10.3 Waiting for the Command to Execute

Monitor the upper two bits (14 & 15) in the \texttt{CDB.StatFlags} field. Until these bits change to report \texttt{PXE\_STATFLAGS\_COMMAND\_COMPLETE} or \texttt{PXE\_STATFLAGS\_COMMAND\_FAILED}, the command has not been executed by the UNDI.

<table>
<thead>
<tr>
<th>StatFlags</th>
<th>Reason</th>
</tr>
</thead>
<tbody>
<tr>
<td>COMMAND_COMPLETE</td>
<td>Command completed successfully. Check StatFlags.</td>
</tr>
<tr>
<td>COMMAND_QUEUED</td>
<td>Command has been queued.</td>
</tr>
<tr>
<td>INITIALIZE</td>
<td>Command has been not executed or queued.</td>
</tr>
</tbody>
</table>

E.4.10.4 Checking Command Execution Results

After command execution completes, either successfully or not, the \texttt{CDB.StatCode} field contains the result of the command execution.

<table>
<thead>
<tr>
<th>StatCode</th>
<th>Reason</th>
</tr>
</thead>
<tbody>
<tr>
<td>SUCCESS</td>
<td>Command completed successfully. Check StatFlags.</td>
</tr>
<tr>
<td>INVALID_CDB</td>
<td>One of the CDB fields was not set correctly.</td>
</tr>
<tr>
<td>BUSY</td>
<td>UNDI is already processing commands. Try again later.</td>
</tr>
<tr>
<td>QUEUE_FULL</td>
<td>Command queue is full. Try again later.</td>
</tr>
<tr>
<td>NOT_STARTED</td>
<td>The UNDI is not started.</td>
</tr>
<tr>
<td>NOT_INITIALIZED</td>
<td>The UNDI is not initialized.</td>
</tr>
</tbody>
</table>
E.4.10.5 StatFlags

If the command was successful, the CDB.StatFlags field reports which external interrupt enable types are currently set. Possible CDB.StatFlags bit settings are:

- PXE_STATFLAGS_INTERRUPT_RECEIVE
- PXE_STATFLAGS_INTERRUPT_TRANSMIT
- PXE_STATFLAGS_INTERRUPT_COMMAND

The bits set in CDB.StatFlags may be different than those that were requested in CDB.OpFlags. For example: If transmit and receive share an external interrupt line, setting either the transmit or receive interrupt will always enable both transmit and receive interrupts. In this case both transmit and receive interrupts will be reported in CDB.StatFlags. Always expect to get more than you ask for!

E.4.11 Receive Filters

This command is used to read and change receive filters and, if supported, read and change the multicast MAC address filter list. Control will not be returned to the caller and the COMMAND_COMPLETE status flag will not be set until the NIC is ready to receive.

E.4.11.1 Issuing the Command

To issue a Receive Filters command, create a CDB and fill it in as shows in the table below:

<table>
<thead>
<tr>
<th>CDB Field</th>
<th>How to initialize the CDB structure for a Receive Filters command</th>
</tr>
</thead>
<tbody>
<tr>
<td>OpCode</td>
<td>PXE_OPCODE_RECEIVE_FILTERS</td>
</tr>
<tr>
<td>OpFlags</td>
<td>Set as needed.</td>
</tr>
<tr>
<td>CPBsize</td>
<td>sizeof(PXE_CPB_RECEIVE_FILTERS)</td>
</tr>
<tr>
<td>DBsize</td>
<td>sizeof(PXE_DB_RECEIVE_FILTERS)</td>
</tr>
<tr>
<td>CPBaddr</td>
<td>Address of PXE_CPB_RECEIVE_FILTERS structure.</td>
</tr>
<tr>
<td>DBaddr</td>
<td>Address of PXE_DB_RECEIVE_FILTERS structure.</td>
</tr>
<tr>
<td>StatCode</td>
<td>PXE_STATCODE_INITIALIZE</td>
</tr>
<tr>
<td>StatFlags</td>
<td>PXE_STATFLAGS_INITIALIZE</td>
</tr>
<tr>
<td>IFnum</td>
<td>A valid interface number from zero to (!PXE.IFcnt</td>
</tr>
<tr>
<td>Control</td>
<td>Set as needed.</td>
</tr>
</tbody>
</table>

E.4.11.2 OpFlags

To read the current receive filter settings set the CDB.OpFlags field to:

- PXE_OPFLAGS_RECEIVE_FILTER_READ

To change the current receive filter settings set one of these OpFlag bits:

- PXE_OPFLAGS_RECEIVE_FILTER_ENABLE
- PXE_OPFLAGS_RECEIVE_FILTER_DISABLE

When changing the receive filter settings, at least one of the OpFlag bits in this list must be selected:
• PXE_OPFLAGS_RECEIVE_FILTER_UNICAST
• PXE_OPFLAGS_RECEIVE_FILTER_BROADCAST
• PXE_OPFLAGS_RECEIVE_FILTER_FILTERED_MULTICAST
• PXE_OPFLAGS_RECEIVE_FILTER_PROMISCUOUS
• PXE_OPFLAGS_RECEIVE_FILTER_ALL_MULTICAST

To clear the contents of the multicast MAC address filter list, set this OpFlag:

• PXE_OPFLAGS_RECEIVE_FILTER_RESET_MCAST_LIST

E.4.11.3 Preparing the CPB

The receive filter CPB is used to change the contents multicast MAC address filter list. To leave the multicast MAC address filter list unchanged, set the CDB.CPBsize field to PXE_CPBSIZE_NOT_USED and CDB.CPBaddr to PXE_CPBADDR_NOT_USED.

To change the multicast MAC address filter list, set CDB.CPBsize to the size, in bytes, of the multicast MAC address filter list and set CDB.CPBaddr to the address of the first entry in the multicast MAC address filter list.

```c
typedef struct s_pxe_cpb_receive_filters {
    // List of multicast MAC addresses. This list, if present,
    // will replace the existing multicast MAC address filter list.
    PXE_MAC_ADDR MCastList[n];
} PXE_CPB_RECEIVE_FILTERS;
```

E.4.11.4 Waiting for the Command to Execute

Monitor the upper two bits (14 & 15) in the CDB.StatFlags field. Until these bits change to report PXE_STATFLAGS_COMMAND_COMPLETE or PXE_STATFLAGS_COMMAND_FAILED, the command has not been executed by the UNDI.

<table>
<thead>
<tr>
<th>StatFlags</th>
<th>Reason</th>
</tr>
</thead>
<tbody>
<tr>
<td>COMMAND_COMPLETE</td>
<td>Command completed successfully. Check StatFlags. DB is written.</td>
</tr>
<tr>
<td>COMMAND_QUEUED</td>
<td>Command has been queued.</td>
</tr>
<tr>
<td>INITIALIZE</td>
<td>Command has been not executed or queued.</td>
</tr>
</tbody>
</table>

E.4.11.5 Checking Command Execution Results

After command execution completes, either successfully or not, the CDB.StatCode field contains the result of the command execution.

<table>
<thead>
<tr>
<th>StatCode</th>
<th>Reason</th>
</tr>
</thead>
<tbody>
<tr>
<td>SUCCESS</td>
<td>Command completed successfully. Check StatFlags. DB is written.</td>
</tr>
<tr>
<td>INVALID_CDB</td>
<td>One of the CDB fields was not set correctly.</td>
</tr>
</tbody>
</table>
E.4.11.6 StatFlags

The receive filter settings in CDB.StatFlags are:

- PXE_STATFLAGS_RECEIVE_FILTER_UNICAST
- PXE_STATFLAGS_RECEIVE_FILTER_BROADCAST
- PXE_STATFLAGS_RECEIVE_FILTER_FILTERED_MULTICAST
- PXE_STATFLAGS_RECEIVE_FILTER_PROMISCUOUS
- PXE_STATFLAGS_RECEIVE_FILTER_ALL_MULTICAST

Unsupported receive filter settings in OpFlags are promoted to the next more liberal receive filter setting. For example: If broadcast or filtered multicast are requested and are not supported by the network device, but promiscuous is; the promiscuous status flag will be set.

E.4.11.7 DB

The DB is used to read the current multicast MAC address filter list. The CDB.DBSIZE and CDB.DBaddr fields can be set to PXE_DBSIZE_NOT_USED and PXE_DBADDR_NOT_USED if the multicast MAC address filter list does not need to be read. When reading the multicast MAC address filter list extra entries in the DB will be filled with zero.

```c
typedef struct s_pxe_db_receive_filters {
    /* Filtered multicast MAC address list. */
    PXE_MAC_ADDR MCastList[n];
} PXE_DB_RECEIVE_FILTERS;
```
E.4.12 Station Address

This command is used to get current station and broadcast MAC addresses and, if supported, to change the current station MAC address.

E.4.12.1 Issuing the Command

To issue a Station Address command, create a CDB and fill it in as shows in the table below:

<table>
<thead>
<tr>
<th>CDB Field</th>
<th>How to initialize the CDB structure for a Station Address command</th>
</tr>
</thead>
<tbody>
<tr>
<td>OpCode</td>
<td>PXE_OPCODE_STATION_ADDRESS</td>
</tr>
<tr>
<td>OpFlags</td>
<td>Set as needed.</td>
</tr>
<tr>
<td>CPBsize</td>
<td>sizeof(PXE_CPB_STATION_ADDRESS)</td>
</tr>
<tr>
<td>DBsize</td>
<td>sizeof(PXE_DB_STATION_ADDRESS)</td>
</tr>
<tr>
<td>CPBaddr</td>
<td>Address of PXE_CPB_STATION_ADDRESS structure.</td>
</tr>
<tr>
<td>DBaddr</td>
<td>Address of PXE_DB_STATION_ADDRESS structure.</td>
</tr>
<tr>
<td>StatCode</td>
<td>PXE_STATCODE_INITIALIZE</td>
</tr>
<tr>
<td>StatFlags</td>
<td>PXE_STATFLAGS_INITIALIZE</td>
</tr>
<tr>
<td>IFnum</td>
<td>A valid interface number from zero to (!PXE.IFcnt</td>
</tr>
<tr>
<td>Control</td>
<td>Set as needed.</td>
</tr>
</tbody>
</table>

E.4.12.2 OpFlags

To read current station and broadcast MAC addresses set the OpFlags field to:

- PXE_OPFLAGS_STATION_ADDRESS_READ

To change the current station to the address given in the CPB set the OpFlags field to:

- PXE_OPFLAGS_STATION_ADDRESS_WRITE

To reset the current station address back to the power on default, set the OpFlags field to:

- PXE_OPFLAGS_STATION_ADDRESS_RESET

E.4.12.3 Preparing the CPB

To change the current station MAC address the CDB.CPBsize and CDB.CPBaddr fields must be set.

```c
typedef struct s_pxe_cpb_station_address {
    // If supplied and supported, the current station MAC address
    // will be changed.

    PXE_MAC_ADDR StationAddr;
} PXE_CPB_STATION_ADDRESS;
```
E.4.12.4 Waiting for the Command to Execute

Monitor the upper two bits (14 & 15) in the CDB.StatFlags field. Until these bits change to report PXE_STATFLAGS_COMMAND_COMPLETE or PXE_STATFLAGS_COMMAND_FAILED, the command has not been executed by the UNDI.

<table>
<thead>
<tr>
<th>StatFlags</th>
<th>Reason</th>
</tr>
</thead>
<tbody>
<tr>
<td>COMMAND_COMPLETE</td>
<td>Command completed successfully. DB is written.</td>
</tr>
<tr>
<td>COMMAND_QUEUED</td>
<td>Command has been queued.</td>
</tr>
<tr>
<td>INITIALIZE</td>
<td>Command has been not executed or queued.</td>
</tr>
</tbody>
</table>

E.4.12.5 Checking Command Execution Results

After command execution completes, either successfully or not, the CDB.StatCode field contains the result of the command execution.

<table>
<thead>
<tr>
<th>StatCode</th>
<th>Reason</th>
</tr>
</thead>
<tbody>
<tr>
<td>SUCCESS</td>
<td>Command completed successfully.</td>
</tr>
<tr>
<td>INVALID_CDB</td>
<td>One of the CDB fields was not set correctly.</td>
</tr>
<tr>
<td>INVALID_CPB</td>
<td>One of the CPB fields was not set correctly.</td>
</tr>
<tr>
<td>BUSY</td>
<td>UNDI is already processing commands. Try again later.</td>
</tr>
<tr>
<td>QUEUE_FULL</td>
<td>Command queue is full. Try again later.</td>
</tr>
<tr>
<td>NOT_STARTED</td>
<td>The UNDI is not started.</td>
</tr>
<tr>
<td>NOT_INITIALIZED</td>
<td>The UNDI is not initialized.</td>
</tr>
<tr>
<td>UNSUPPORTED</td>
<td>The requested operation is not supported.</td>
</tr>
</tbody>
</table>

E.4.12.6 Before Using the DB

The DB is used to read the current station, broadcast and permanent station MAC addresses. The CDB.DBsize and CDB.DBaddr fields can be set to PXE_DBSIZE_NOT_USED and PXE_DBADDR_NOT_USED if these addresses do not need to be read.

```c
typedef struct s_pxe_db_station_address {
    // Current station MAC address.
    PXE_MAC_ADDR StationAddr;

    // Station broadcast MAC address.
    PXE_MAC_ADDR BroadcastAddr;

    // Permanent station MAC address.
    PXE_MAC_ADDR PermanentAddr;
} PXE_DB_STATION_ADDRESS;
```
E.4.13 Statistics

This command is used to read and clear the NIC traffic statistics. Before using this command check to see if statistics is supported in the `!PXE.Implementation` flags.

E.4.13.1 Issuing the Command

To issue a Statistics command, create a CDB and fill it in as shown in the table below:

<table>
<thead>
<tr>
<th>CDB Field</th>
<th>How to initialize the CDB structure for a Statistics command</th>
</tr>
</thead>
<tbody>
<tr>
<td>OpCode</td>
<td>PXE_OPCODE_STATISTICS</td>
</tr>
<tr>
<td>OpFlags</td>
<td>Set as needed.</td>
</tr>
<tr>
<td>CPBsize</td>
<td>PXE_CPBSIZE_NOT_USED</td>
</tr>
<tr>
<td>DBsize</td>
<td>sizeof(PXE_DB_STATISTICS)</td>
</tr>
<tr>
<td>CPBaddr</td>
<td>PXE_CPBADDR_NOT_USED</td>
</tr>
<tr>
<td>DBaddr</td>
<td>Address of PXE_DB_STATISTICS structure.</td>
</tr>
<tr>
<td>StatCode</td>
<td>PXE_STATCODE_INITIALIZE</td>
</tr>
<tr>
<td>StatFlags</td>
<td>PXE_STATFLAGS_INITIALIZE</td>
</tr>
<tr>
<td>IFnum</td>
<td>A valid interface number from zero to (!PXE.IFcnt</td>
</tr>
<tr>
<td>Control</td>
<td>Set as needed.</td>
</tr>
</tbody>
</table>

E.4.13.2 OpFlags

To read the current statistics counters set the OpFlags field to:

`PXE_OPFLAGS_STATISTICS_READ`

To reset the current statistics counters set the OpFlags field to:

`PXE_OPFLAGS_STATISTICS_RESET`

E.4.13.3 Waiting for the Command to Execute

Monitor the upper two bits (14 & 15) in the CDB.StatFlags field. Until these bits change to report PXE_STATFLAGS_COMMAND_COMPLETE or PXE_STATFLAGS_COMMAND_FAILED, the command has not been executed by the UNDI.

<table>
<thead>
<tr>
<th>StatFlags</th>
<th>Reason</th>
</tr>
</thead>
<tbody>
<tr>
<td>COMMAND_COMPLETE</td>
<td>Command completed successfully. DB is written.</td>
</tr>
<tr>
<td>COMMAND_QUEUED</td>
<td>Command has been queued.</td>
</tr>
<tr>
<td>INITIALIZE</td>
<td>Command has been not executed or queued.</td>
</tr>
</tbody>
</table>
E.4.13.4 Checking Command Execution Results

After command execution completes, either successfully or not, the `CDB.StatCode` field contains the result of the command execution.

<table>
<thead>
<tr>
<th>StatCode</th>
<th>Reason</th>
</tr>
</thead>
<tbody>
<tr>
<td>SUCCESS</td>
<td>Command completed successfully. DB is written.</td>
</tr>
<tr>
<td>INVALID_CDB</td>
<td>One of the CDB fields was not set correctly.</td>
</tr>
<tr>
<td>BUSY</td>
<td>UNDI is already processing commands. Try again later.</td>
</tr>
<tr>
<td>QUEUE_FULL</td>
<td>Command queue is full. Try again later.</td>
</tr>
<tr>
<td>NOT_STARTED</td>
<td>The UNDI is not started.</td>
</tr>
<tr>
<td>NOT_INITIALIZED</td>
<td>The UNDI is not initialized.</td>
</tr>
<tr>
<td>UNSUPPORTED</td>
<td>This command is not supported.</td>
</tr>
</tbody>
</table>

E.4.13.5 DB

Unsupported statistics counters will be zero filled by UNDI.

```c
typedef struct s_pxe_db_statistics {
    // Bit field identifying what statistic data is collected by
    // the UNDI/NIC.
    // If bit 0x00 is set, Data[0x00] is collected.
    // If bit 0x01 is set, Data[0x01] is collected.
    // If bit 0x20 is set, Data[0x20] is collected.
    // If bit 0x21 is set, Data[0x21] is collected.
    // Etc.
    PXE_UINT64 Supported;
    // Statistic data.
    PXE_UINT64 Data[64];
} PXE_DB_STATISTICS;
```

- `#define PXE_STATISTICS_RX_TOTAL_FRAMES   0x00`
  // Total number of frames received. Includes frames with errors
  // and dropped frames.
- `#define PXE_STATISTICS_RX_GOOD_FRAMES    0x01`
  // Number of valid frames received and copied into receive
  // buffers.
- `#define PXE_STATISTICS_RX_UNDERSIZE_FRAMES   0x02`
  // Number of frames below the minimum length for the media.
  // This would be <64 for ethernet.
- `#define PXE_STATISTICS_RX_OVERSIZE_FRAMES   0x03`
  // Number of frames longer than the maximum length for the
  // media. This would be >1500 for ethernet.

#define PXE_STATISTICS_RX_OVERSIZE_FRAMES 0x03

// Valid frames that were dropped because receive buffers
// were full.
#define PXE_STATISTICS_RX_DROPPED_FRAMES 0x04

// Number of valid unicast frames received and not dropped.
#define PXE_STATISTICS_RX_UNICAST_FRAMES 0x05

// Number of valid broadcast frames received and not dropped.
#define PXE_STATISTICS_RX_BROADCAST_FRAMES 0x06

// Number of valid multicast frames received and not dropped.
#define PXE_STATISTICS_RX_MULTICAST_FRAMES 0x07

// Number of frames w/ CRC or alignment errors.
#define PXE_STATISTICS_RX_CRC_ERROR_FRAMES 0x08

// Total number of bytes received. Includes frames with errors
// and dropped frames.
#define PXE_STATISTICS_RX_TOTAL_BYTES 0x09

// Transmit statistics.
#define PXE_STATISTICS_TX_TOTAL_FRAMES 0xA
#define PXE_STATISTICS_TX_GOOD_FRAMES 0xB
#define PXE_STATISTICS_TX_UNDERSIZE_FRAMES 0xC
#define PXE_STATISTICS_TX_OVERSIZE_FRAMES 0xD
#define PXE_STATISTICS_TX_DROPPED_FRAMES 0xE
#define PXE_STATISTICS_TX_UNICAST_FRAMES 0xF
#define PXE_STATISTICS_TX_BROADCAST_FRAMES 0x10
#define PXE_STATISTICS_TX_MULTICAST_FRAMES 0x11
#define PXE_STATISTICS_TX_CRC_ERROR_FRAMES 0x12
#define PXE_STATISTICS_TX_TOTAL_BYTES 0x13

// Number of collisions detection on this subnet.
#define PXE_STATISTICS_COLLISIONS 0x14

// Number of frames destined for unsupported protocol.
#define PXE_STATISTICS_UNSUPPORTED_PROTOCOL 0x15

// Number of valid frames received that were duplicated.
#define PXE_STATISTICS_RX_DUPLICATED_FRAMES 0x16

// Number of encrypted frames received that failed to decrypt.
#define PXE_STATISTICS_RX_DECRYPT_ERROR_FRAMES 0x17

// Number of frames that failed to transmit after exceeding the
// retry limit.
#define PXE_STATISTICS_TX_ERROR_FRAMES 0x18

// Number of frames transmitted successfully after more than one
// attempt.
#define PXE_STATISTICS_TX_RETRY_FRAMES 0x19

E.4.14 MCast IP To MAC

Translate a multicast IPv4 or IPv6 address to a multicast MAC address.

E.4.14.1 Issuing the Command

To issue a MCast IP To MAC command, create a CDB and fill it in as shown in the table below:

<table>
<thead>
<tr>
<th>CDB Field</th>
<th>How to initialize the CDB structure for a MCast IP To MAC command</th>
</tr>
</thead>
<tbody>
<tr>
<td>OpCode</td>
<td>PXE_OPCODE_MCAST_IP_TO_MAC</td>
</tr>
<tr>
<td>OpFlags</td>
<td>Set as needed.</td>
</tr>
<tr>
<td>CPBsize</td>
<td>sizeof(PXE_CPB_MCAST_IP_TO_MAC)</td>
</tr>
<tr>
<td>DBsize</td>
<td>sizeof(PXE_DB_MCAST_IP_TO_MAC)</td>
</tr>
<tr>
<td>CPBaddr</td>
<td>Address of PXE_CPB_MCAST_IP_TO_MAC structure.</td>
</tr>
<tr>
<td>Dbaddr</td>
<td>Address of PXE_DB_MCAST_IP_TO_MAC structure.</td>
</tr>
<tr>
<td>StatCode</td>
<td>PXE_STATCODE_INITIALIZE</td>
</tr>
<tr>
<td>StatFlags</td>
<td>PXE_STATFLAGS_INITIALIZE</td>
</tr>
<tr>
<td>Ifnum</td>
<td>A valid interface number from zero to !PXE.IFcnt</td>
</tr>
<tr>
<td>Control</td>
<td>Set as needed.</td>
</tr>
</tbody>
</table>

E.4.14.2 OpFlags

To convert a multicast IP address to a multicast MAC address the UNDI needs to know the format of the IP address. Set one of these OpFlags to identify the format of the IP addresses in the CPB:

PXE_OPFLAGS_MCAST_IPV4_TO_MAC
PXE_OPFLAGS_MCAST_IPV6_TO_MAC

E.4.14.3 Preparing the CPB

Fill in an array of one or more multicast IP addresses. Be sure to set the CDB.CPBsize and CDB.CPBaddr fields accordingly.

typedef struct s_pxe_cpb_mcast_ip_to_mac {

    // Multicast IP address to be converted to multicast
    // MAC address.
    PXE_IP_ADDR IP[n];
} PXE_CPB_MCAST_IP_TO_MAC;
E.4.14.4 Waiting for the Command to Execute

Monitor the upper two bits (14 & 15) in the CDB.StatFlags field. Until these bits change to report PXE_STATFLAGS_COMMAND_COMPLETE or PXE_STATFLAGS_COMMAND_FAILED, the command has not been executed by the UNDI.

<table>
<thead>
<tr>
<th>StatFlags</th>
<th>Reason</th>
</tr>
</thead>
<tbody>
<tr>
<td>COMMAND_COMPLETE</td>
<td>Command completed successfully. DB is written.</td>
</tr>
<tr>
<td>COMMAND_QUEUED</td>
<td>Command has been queued.</td>
</tr>
<tr>
<td>INITIALIZE</td>
<td>Command has not executed or queued.</td>
</tr>
</tbody>
</table>

E.4.14.5 Checking Command Execution Results

After command execution completes, either successfully or not, the CDB.StatCode field contains the result of the command execution.

<table>
<thead>
<tr>
<th>StatCode</th>
<th>Reason</th>
</tr>
</thead>
<tbody>
<tr>
<td>SUCCESS</td>
<td>Command completed successfully. DB is written.</td>
</tr>
<tr>
<td>INVALID_CDB</td>
<td>One of the CDB fields was not set correctly.</td>
</tr>
<tr>
<td>INVALID_CPB</td>
<td>One of the CPB fields was not set correctly.</td>
</tr>
<tr>
<td>BUSY</td>
<td>UNDI is already processing commands. Try again later.</td>
</tr>
<tr>
<td>QUEUE_FULL</td>
<td>Command queue is full. Try again later.</td>
</tr>
<tr>
<td>NOT_STARTED</td>
<td>The UNDI is not started.</td>
</tr>
<tr>
<td>NOT_INITIALIZED</td>
<td>The UNDI is not initialized.</td>
</tr>
</tbody>
</table>

E.4.14.6 Before Using the DB

The DB is where the multicast MAC addresses will be written.

```c
typedef struct s_pxe_db_mcast_ip_to_mac {
    // Multicast MAC address.
    PXE_MAC_ADDR MAC[n];
} PXE_DB_MCAST_IP_TO_MAC;
```

E.4.15 NvData

This command is used to read and write (if supported by NIC H/W) nonvolatile storage on the NIC. Nonvolatile storage could be EEPROM, FLASH or battery backed RAM.
E.4.15.1 Issuing the Command

To issue a NvData command, create a CDB and fill it in as shown in the table below:

<table>
<thead>
<tr>
<th>CDB Field</th>
<th>How to initialize the CDB structure for a NvData command</th>
</tr>
</thead>
<tbody>
<tr>
<td>OpCode</td>
<td>PXE_OPCODE_NVDATA</td>
</tr>
<tr>
<td>OpFlags</td>
<td>Set as needed.</td>
</tr>
<tr>
<td>CPBsize</td>
<td>sizeof(PXE_CPB_NVDATA)</td>
</tr>
<tr>
<td>DBsize</td>
<td>sizeof(PXE_DB_NVDATA)</td>
</tr>
<tr>
<td>CPBaddr</td>
<td>Address of PXE_CPB_NVDATA structure.</td>
</tr>
<tr>
<td>Dbaddr</td>
<td>Address of PXE_DB_NVDATA structure.</td>
</tr>
<tr>
<td>StatCode</td>
<td>PXE_STATCODE_INITIALIZE</td>
</tr>
<tr>
<td>StatFlags</td>
<td>PXE_STATFLAGS_INITIALIZE</td>
</tr>
</tbody>
</table>
| Ifnum       | A valid interface number from zero to 

\(!PXE.IFcnt | (!PXE.IFcntExt << 8))

| Control     | Set as needed.                                           |

E.4.15.2 Preparing the CPB

There are two types of nonvolatile data CPBs, one for sparse updates and one for bulk updates. Sparse updates allow updating of single nonvolatile storage items. Bulk updates always update all nonvolatile storage items. Check the \!PXE.Implementation flags to see which type of nonvolatile update is supported by this UNDI and network device.

If you do not need to update the nonvolatile storage set the CDB.CPBsize and CDB.CPBaddr fields to PXE_CPBSIZE_NOT_USED and PXE_CPBADDR_NOT_USED.

E.4.15.2.1 Sparse NvData CPB

typedef struct s_pxe_cpb_nvdata_sparse {
    // NvData item list. Only items in this list will be updated.

    struct {

        // Nonvolatile storage address to be changed.
        PXE_UINT32 Addr;

        // Data item to write into above storage address.
        union {
            PXE_UINT8 Byte;
            PXE_UINT16 Word;
            PXE_UINT32 Dword;
        } Data;
    } Item[n];

    } PXE_CPB_NVDATA SPARSE;
E.4.15.2.2 Bulk NvData CPB

// When using bulk update, the size of the CPB structure must be
// the same size as the nonvolatile NIC storage.

typedef union u_pxe_cpb_nvdata_bulk {

    // Array of byte-wide data items.
    PXE_UINT8 Byte[n];

    // Array of word-wide data items.
    PXE_UINT16 Word[n];

    // Array of dword-wide data items.
    PXE_UINT32 Dword[n];
} PXE_CPB_NVDATA_BULK;

E.4.15.3 Waiting for the Command to Execute

Monitor the upper two bits (14 & 15) in the CDB.StatFlags field. Until these bits change to report PXE_STATFLAGS_COMMAND_COMPLETE or PXE_STATFLAGS_COMMAND_FAILED, the command has not been executed by the UNDI.

<table>
<thead>
<tr>
<th>StatFlags</th>
<th>Reason</th>
</tr>
</thead>
<tbody>
<tr>
<td>COMMAND_COMPLETE</td>
<td>Command completed successfully. Nonvolatile data is updated from CPB and/or written to DB.</td>
</tr>
<tr>
<td>COMMAND_QUEUED</td>
<td>Command has been queued.</td>
</tr>
<tr>
<td>INITIALIZE</td>
<td>Command has been not executed or queued.</td>
</tr>
</tbody>
</table>

E.4.15.4 Checking Command Execution Results

After command execution completes, either successfully or not, the CDB.StatCode field contains the result of the command execution.

<table>
<thead>
<tr>
<th>StatCode</th>
<th>Reason</th>
</tr>
</thead>
<tbody>
<tr>
<td>SUCCESS</td>
<td>Command completed successfully. Nonvolatile data is updated from CPB and/or written to DB.</td>
</tr>
<tr>
<td>INVALID_CDB</td>
<td>One of the CDB fields was not set correctly.</td>
</tr>
<tr>
<td>INVALID_CPB</td>
<td>One of the CPB fields was not set correctly.</td>
</tr>
<tr>
<td>BUSY</td>
<td>UNDI is already processing commands. Try again later.</td>
</tr>
<tr>
<td>QUEUE FULL</td>
<td>Command queue is full. Try again later.</td>
</tr>
<tr>
<td>NOT_STARTED</td>
<td>The UNDI is not started.</td>
</tr>
<tr>
<td>NOT_INITIALIZED</td>
<td>The UNDI is not initialized.</td>
</tr>
<tr>
<td>UNSUPPORTED</td>
<td>Requested operation is unsupported.</td>
</tr>
</tbody>
</table>
E.4.15.4.1 DB
Check the width and number of nonvolatile storage items. This information is returned by the Get Init Info command.

```c
typedef struct s_pxe_db_nvdata {
    // Arrays of data items from nonvolatile storage.
    union {
        // Array of byte-wide data items.
        PXE_UINT8 Byte[n];
        // Array of word-wide data items.
        PXE_UINT16 Word[n];
        // Array of dword-wide data items.
        PXE_UINT32 Dword[n];
    } Data;
} PXE_DB_NVDATA;
```

E.4.16 Get Status
This command returns the current interrupt status and/or the transmitted buffer addresses and the current media status. If the current interrupt status is returned, pending interrupts will be acknowledged by this command. Transmitted buffer addresses that are written to the DB are removed from the transmitted buffer queue.

This command may be used in a polled fashion with external interrupts disabled.

E.4.16.1 Issuing the Command
To issue a Get Status command, create a CDB and fill it in as shown in the table below:

<table>
<thead>
<tr>
<th>CDB Field</th>
<th>How to initialize the CDB structure for a Get Status command</th>
</tr>
</thead>
<tbody>
<tr>
<td>OpCode</td>
<td>PXE_OPCODE_GET_STATUS</td>
</tr>
<tr>
<td>OpFlags</td>
<td>Set as needed.</td>
</tr>
<tr>
<td>CPBsize</td>
<td>PXE_CPBSIZE_NOT_USED</td>
</tr>
<tr>
<td>DBsize</td>
<td>Sizeof(PXE_DB_GET_STATUS)</td>
</tr>
<tr>
<td>CPBaddr</td>
<td>PXE_CPBADDR_NOT_USED</td>
</tr>
<tr>
<td>DBaddr</td>
<td>Address of PXE_DB_GET_STATUS structure.</td>
</tr>
<tr>
<td>StatCode</td>
<td>PXE_STATCODE_INITIALIZE</td>
</tr>
<tr>
<td>StatFlags</td>
<td>PXE_STATFLAGS_INITIALIZE</td>
</tr>
<tr>
<td>IFnum</td>
<td>A valid interface number from zero to (!PXE.IFcnt</td>
</tr>
<tr>
<td>Control</td>
<td>Set as needed.</td>
</tr>
</tbody>
</table>
E.4.16.1.1 Setting OpFlags
Set one or a combination of the OpFlags below to return the interrupt status and/or the transmitted buffer addresses and/or the media status.

- PXE_OPFLAGS_GET_INTERRUPT_STATUS
- PXE_OPFLAGS_GET_TRANSMITTED_BUFFERS
- PXE_OPFLAGS_GET_MEDIA_STATUS

E.4.16.2 Waiting for the Command to Execute
Monitor the upper two bits (14 & 15) in the CDB.StatFlags field. Until these bits change to report PXE_STATFLAGS_COMMAND_COMPLETE or PXE_STATFLAGS_COMMAND_FAILED, the command has not been executed by the UNDI.

<table>
<thead>
<tr>
<th>StatFlags</th>
<th>Reason</th>
</tr>
</thead>
<tbody>
<tr>
<td>COMMAND_COMPLETE</td>
<td>Command completed successfully. StatFlags and/or DB are updated.</td>
</tr>
<tr>
<td>COMMAND_QUEUED</td>
<td>Command has been queued.</td>
</tr>
<tr>
<td>INITIALIZE</td>
<td>Command has been not executed or queued.</td>
</tr>
</tbody>
</table>

E.4.16.3 Checking Command Execution Results
After command execution completes, either successfully or not, the CDB.StatCode field contains the result of the command execution.

<table>
<thead>
<tr>
<th>StatCode</th>
<th>Reason</th>
</tr>
</thead>
<tbody>
<tr>
<td>SUCCESS</td>
<td>Command completed successfully. StatFlags and/or DB are updated.</td>
</tr>
<tr>
<td>INVALID_CDB</td>
<td>One of the CDB fields was not set correctly.</td>
</tr>
<tr>
<td>BUSY</td>
<td>UNDI is already processing commands. Try again later.</td>
</tr>
<tr>
<td>QUEUE_FULL</td>
<td>Command queue is full. Try again later.</td>
</tr>
<tr>
<td>NOT_STARTED</td>
<td>The UNDI is not started.</td>
</tr>
<tr>
<td>NOT_INITIALIZED</td>
<td>The UNDI is not initialized.</td>
</tr>
</tbody>
</table>

E.4.16.4 StatFlags
If the command completes successfully and the PXE_OPFLAGS_GET_INTERRUPT_STATUS OpFlag was set in the CDB, the current interrupt status is returned in the CDB.StatFlags field and any pending interrupts will have been cleared.

- PXE_STATFLAGS_GET_STATUS_RECEIVE
- PXE_STATFLAGS_GET_STATUS_TRANSMIT
- PXE_STATFLAGS_GET_STATUS_COMMAND
- PXE_STATFLAGS_GET_STATUS_SOFTWARE

The StatFlags above may not map directly to external interrupt signals. For example: Some NICs may combine both the receive and transmit interrupts to one external interrupt line. When a receive and/or transmit interrupt occurs, use the Get Status to determine which type(s) of interrupt(s) occurred.
This flag is set if the transmitted buffer queue is empty. This flag will be set if all transmitted buffer addresses get written into the DB.

PXE_STATFLAGS_GET_STATUS_TXBUF_QUEUE_EMPTY

This flag is set if no transmitted buffer addresses were written into the DB.

PXE_STATFLAGS_GET_STATUS_NO_TXBUFS_WRITTEN

This flag is set if there is no media present.

PXE_STATFLAGS_GET_STATUS_NO_MEDIA

### E.4.16.5 Using the DB

When reading the transmitted buffer addresses there should be room for at least one 64-bit address in the DB. Once a complete transmitted buffer address is written into the DB, the address is removed from the transmitted buffer queue. If the transmitted buffer queue is full, attempts to use the Transmit command will fail.

```c
#pragma pack(1)
typedef struct s_pxe_db_get_status {
    // Length of next receive frame (header + data). If this is zero, there is no next receive frame available.
    PXE_UINT32    RxFrameLen;
    // Reserved, set to zero.
    PXE_UINT32    reserved;
    // Addresses of transmitted buffers that need to be recycled.
    PXE_UINT64    xBuffer[n];
} PXE_DB_GET_STATUS;
#pragma pack()
```

### E.4.17 Fill Header

This command is used to fill the media header(s) in transmit packet(s).
E.4.17.1 Issuing the Command

To issue a Fill Header command, create a CDB and fill it in as shown in the table below:

<table>
<thead>
<tr>
<th>CDB Field</th>
<th>How to initialize the CDB structure for a Fill Header command</th>
</tr>
</thead>
<tbody>
<tr>
<td>OpCode</td>
<td>PXE_OPCODE_FILL_HEADER</td>
</tr>
<tr>
<td>OpFlags</td>
<td>Set as needed.</td>
</tr>
<tr>
<td>CPBsize</td>
<td>PXE_CPB_FILL_HEADER</td>
</tr>
<tr>
<td>DBsize</td>
<td>PXE_DBSIZE_NOT_USED</td>
</tr>
<tr>
<td>CPBaddr</td>
<td>Address of a PXE_CPB_FILL_HEADER structure.</td>
</tr>
<tr>
<td>DBaddr</td>
<td>PXE_DBADDR_NOT_USED</td>
</tr>
<tr>
<td>StatCode</td>
<td>PXE_STATCODE_INITIALIZE</td>
</tr>
<tr>
<td>StatFlags</td>
<td>PXE_STATFLAGS_INITIALIZE</td>
</tr>
<tr>
<td>IFnum</td>
<td>A valid interface number from zero to (!PXE.IFcnt</td>
</tr>
<tr>
<td>Control</td>
<td>Set as needed.</td>
</tr>
</tbody>
</table>

E.4.17.2 OpFlags

Select one of the OpFlags below so the UNDI knows what type of CPB is being used.

- PXE_OPFLAGS_FILL_HEADER_WHOLE
- PXE_OPFLAGS_FILL_HEADER_FRAGMENTED

E.4.17.3 Preparing the CPB

If multiple frames per command are supported (see !PXE.Implementation flags), multiple CPBs can be packed together. The CDB.CPBsize field lets the UNDI know how many CPBs are packed together.

E.4.17.4 Nonfragmented Frame

```c
#pragma pack(1)
typedef struct s_pxe_cpb_fill_header {

    // Source and destination MAC addresses. These will be copied
    // into the media header without doing byte swapping.
    PXE_MAC_ADDR SrcAddr;
    PXE_MAC_ADDR DestAddr;

    // Address of first byte of media header. The first byte of
    // packet data follows the last byte of the media header.
    PXE_UINT64 MediaHeader;

    // Length of packet data in bytes (not including the media
    // header).
    PXE_UINT32 PacketLen;
}
```
// Protocol type. This will be copied into the media header
// without doing byte swapping. Protocol type numbers can be
// obtained from the Assigned Numbers RFC 3232.
PXU16 Protocol;

// Length of the media header in bytes.
PXE_UINT16 MediaHeaderLen;
}

#pragma pack()
#define PXE_PROTOCOL_ETHERNET_IP 0x0800
#define PXE_PROTOCOL_ETHERNET_ARP 0x0806

E.4.17.5 Fragmented Frame
#pragma pack(1)
typedef struct s_pxe_cpb_fill_header_fragmented {

    // Source and destination MAC addresses. These will be copied
    // into the media header without doing byte swapping.
    PXE_MAC_ADDR SrcAddr;
PXE_MAC_ADDR DestAddr;

    // Length of packet data in bytes (not including the media
    // header).
    PXE_UINT32 PacketLen;

    // Protocol type. This will be copied into the media header
    // without doing byte swapping. Protocol type numbers can be
    // obtained from the Assigned Numbers RFC 3232.
    PXE_MEDIA_PROTOCOL Protocol;

    // Length of the media header in bytes.
PXE_UINT16 MediaHeaderLen;

    // Number of packet fragment descriptors.
PXE_UINT16 FragCnt;

    // Reserved, must be set to zero.
PXE_UINT16 reserved;

    // Array of packet fragment descriptors. The first byte of the
    // media header is the first byte of the first fragment.
    struct {

        // Address of this packet fragment.
        PXE_UINT64 FragAddr;

```
E.4.17.6 Waiting for the Command to Execute

Monitor the upper two bits (14 & 15) in the `CDB.StatFlags` field. Until these bits change to report `PXE_STATFLAGS_COMMAND_COMPLETE` or `PXE_STATFLAGS_COMMAND_FAILED`, the command has not been executed by the UNDI.

<table>
<thead>
<tr>
<th>StatFlags</th>
<th>Reason</th>
</tr>
</thead>
<tbody>
<tr>
<td>COMMAND_COMPLETE</td>
<td>Command completed successfully. Frame is ready to transmit.</td>
</tr>
<tr>
<td>COMMAND_QUEUED</td>
<td>Command has been queued.</td>
</tr>
<tr>
<td>INITIALIZE</td>
<td>Command has been not executed or queued.</td>
</tr>
</tbody>
</table>

E.4.17.7 Checking Command Execution Results

After command execution completes, either successfully or not, the `CDB.StatCode` field contains the result of the command execution.

<table>
<thead>
<tr>
<th>StatCode</th>
<th>Reason</th>
</tr>
</thead>
<tbody>
<tr>
<td>SUCCESS</td>
<td>Command completed successfully. Frame is ready to transmit.</td>
</tr>
<tr>
<td>INVALID_CDB</td>
<td>One of the CDB fields was not set correctly.</td>
</tr>
<tr>
<td>INVALID_CPB</td>
<td>One of the CPB fields was not set correctly.</td>
</tr>
<tr>
<td>BUSY</td>
<td>UNDI is already processing commands. Try again later.</td>
</tr>
<tr>
<td>QUEUE_FULL</td>
<td>Command queue is full. Try again later.</td>
</tr>
<tr>
<td>NOT_STARTED</td>
<td>The UNDI is not started.</td>
</tr>
<tr>
<td>NOT_INITIALIZED</td>
<td>The UNDI is not initialized.</td>
</tr>
</tbody>
</table>
E.4.18 Transmit

The Transmit command is used to place a packet into the transmit queue. The data buffers given to this command are to be considered locked and the application or universal network driver loses the ownership of those buffers and must not free or relocate them until the ownership returns.

When the packets are transmitted, a transmit complete interrupt is generated (if interrupts are disabled, the transmit interrupt status is still set and can be checked using the Get Status command).

Some UNDI implementations and network adapters support transmitting multiple packets with one transmit command. If this feature is supported, multiple transmit CPBs can be linked in one transmit command.

Though all UNDIs support fragmented frames, the same cannot be said for all network devices or protocols. If a fragmented frame CPB is given to UNDI and the network device does not support fragmented frames (see PXE.Implementation flags), the UNDI will have to copy the fragments into a local buffer before transmitting.

E.4.18.1 Issuing the Command

To issue a Transmit command, create a CDB and fill it in as shown in the table below:

<table>
<thead>
<tr>
<th>CDB Field</th>
<th>How to initialize the CDB structure for a Transmit command</th>
</tr>
</thead>
<tbody>
<tr>
<td>OpCode</td>
<td>PXE_OPCODE_TRANSMIT</td>
</tr>
<tr>
<td>OpFlags</td>
<td>Set as needed.</td>
</tr>
<tr>
<td>CPBsize</td>
<td>sizeof(PXE_CPB_TRANSMIT)</td>
</tr>
<tr>
<td>DBsize</td>
<td>PXE_DBSIZE_NOT_USED</td>
</tr>
<tr>
<td>CPBaddr</td>
<td>Address of a PXE_CPB_TRANSMIT structure.</td>
</tr>
<tr>
<td>DBaddr</td>
<td>PXE_DBADDR_NOT_USED</td>
</tr>
<tr>
<td>StatCode</td>
<td>PXE_STATCODE_INITIALIZE</td>
</tr>
<tr>
<td>StatFlags</td>
<td>PXE_STATFLAGS_INITIALIZE</td>
</tr>
<tr>
<td>IFnum</td>
<td>A valid interface number from zero to (!PXE.IFcnt</td>
</tr>
<tr>
<td>Control</td>
<td>Set as needed.</td>
</tr>
</tbody>
</table>
E.4.18.2 OpFlags

Check the !PXE.Implementation flags to see if the network device support fragmented packets. Select one of the OpFlags below so the UNDI knows what type of CPB is being used.

- PXE_OPFLAGS_TRANSMIT_WHOLE
- PXE_OPFLAGS_TRANSMIT_FRAGMENTED

In addition to selecting whether or not fragmented packets are being given, S/W UNDI needs to know if it should block until the packets are transmitted. H/W UNDI cannot block, these two OpFlag settings have no affect when used with H/W UNDI.

- PXE_OPFLAGS_TRANSMIT_BLOCK
- PXE_OPFLAGS_TRANSMIT_DONT_BLOCK

E.4.18.3 Preparing the CPB

If multiple frames per command are supported (see !PXE.Implementation flags), multiple CPBs can be packed together. The CDB.CPBsize field lets the UNDI know how many frames are to be transmitted.

E.4.18.4 Nonfragmented Frame

```c
#pragma pack(1)
typedef struct s_pxe_cpb_transmit {
    // Address of first byte of frame buffer. This is also the first byte of the media header. This address must be a processor-based address for S/W UNDI and a device-based address for H/W UNDI.
PXE_UINT64 FrameAddr;

    // Length of the data portion of the frame buffer in bytes. Do not include the length of the media header.
PXE_UINT32 DataLen;

    // Length of the media header in bytes.
PXE_UINT16 MediaheaderLen;

    // Reserved, must be zero.
PXE_UINT16 reserved;
} PXE_CPB_TRANSMIT;
#pragma pack()
```

E.4.18.5 Fragmented Frame

```c
#pragma pack(1)
typedef struct s_pxe_cpb_transmit_fragments {
    // Length of packet data in bytes (not including the media header).
PXE_UINT32 FrameLen;
```
// Length of the media header in bytes.
PXE_UINT16 MediaheaderLen;

// Number of packet fragment descriptors.
PXE_UINT16 FragCnt;

// Array of frame fragment descriptors. The first byte of the
// first fragment is also the first byte of the media header.
struct {
// Address of this frame fragment. This address must be a
// processor-based address for S/W UNDI and a device-based
// address for H/W UNDI.
  PXE_UINT64 FragAddr;

  // Length of this frame fragment.
  PXE_UINT32 FragLen;

  // Reserved, must be set to zero.
  PXE_UINT32 reserved;
} FragDesc[n];
} PXE_CPB_TRANSMIT_FRAGMENTS;
#pragma pack()

E.4.18.6 Waiting for the Command to Execute

Monitor the upper two bits (14 & 15) in the CDB.StatFlags field. Until these bits change to report
PXE_STATFLAGS_COMMAND_COMPLETE or PXE_STATFLAGS_COMMAND_FAILED, the command has not
been executed by the UNDI.

<table>
<thead>
<tr>
<th>StatFlags</th>
<th>Reason</th>
</tr>
</thead>
<tbody>
<tr>
<td>COMMAND_COMPLETE</td>
<td>Command completed successfully. Use the Get Status command to see when frame buffers can be reused.</td>
</tr>
<tr>
<td>COMMAND_QUEUED</td>
<td>Command has been queued.</td>
</tr>
<tr>
<td>INITIALIZE</td>
<td>Command has been not executed or queued.</td>
</tr>
</tbody>
</table>
E.4.18.7 Checking Command Execution Results

After command execution completes, either successfully or not, the CDB.StatCode field contains the result of the command execution.

<table>
<thead>
<tr>
<th>StatCode</th>
<th>Reason</th>
</tr>
</thead>
<tbody>
<tr>
<td>SUCCESS</td>
<td>Command completed successfully. Use the Get Status command to see when frame buffers can be reused.</td>
</tr>
<tr>
<td>INVALID_CDB</td>
<td>One of the CDB fields was not set correctly.</td>
</tr>
<tr>
<td>INVALID_CPB</td>
<td>One of the CPB fields was not set correctly.</td>
</tr>
<tr>
<td>BUSY</td>
<td>UNDI is already processing commands. Try again later.</td>
</tr>
<tr>
<td>QUEUE_FULL</td>
<td>Command queue is full. Wait for queued commands to complete. Try again later.</td>
</tr>
<tr>
<td>BUFFER_FULL</td>
<td>Transmit buffer is full. Call Get Status command to empty buffer.</td>
</tr>
<tr>
<td>NOT_STARTED</td>
<td>The UNDI is not started.</td>
</tr>
<tr>
<td>NOT_INITIALIZED</td>
<td>The UNDI is not initialized.</td>
</tr>
</tbody>
</table>

E.4.19 Receive

When the network adapter has received a frame, this command is used to copy the frame into driver/application storage. Once a frame has been copied, it is removed from the receive queue.

E.4.19.1 Issuing the Command

To issue a Receive command, create a CDB and fill it in as shown in the table below:

<table>
<thead>
<tr>
<th>CDB Field</th>
<th>How to initialize the CDB structure for a Receive command</th>
</tr>
</thead>
<tbody>
<tr>
<td>OpCode</td>
<td>PXE_OPCODE_RECEIVE</td>
</tr>
<tr>
<td>OpFlags</td>
<td>Set as needed.</td>
</tr>
<tr>
<td>CPBsize</td>
<td>sizeof(PXE_CPB_RECEIVE)</td>
</tr>
<tr>
<td>DBsize</td>
<td>sizeof(PXE_DB_RECEIVE)</td>
</tr>
<tr>
<td>CPBaddr</td>
<td>Address of a PXE_CPB_RECEIVE structure.</td>
</tr>
<tr>
<td>DBaddr</td>
<td>Address of a PXE_DB_RECEIVE structure.</td>
</tr>
<tr>
<td>StatCode</td>
<td>PXE_STATCODE_INITIALIZE</td>
</tr>
<tr>
<td>StatFlags</td>
<td>PXE_STATFLAGS_INITIALIZE</td>
</tr>
<tr>
<td>Ifnum</td>
<td>A valid interface number from zero to (!PXE.IFcnt</td>
</tr>
<tr>
<td>Control</td>
<td>Set as needed.</td>
</tr>
</tbody>
</table>

E.4.19.2 Preparing the CPB

If multiple frames per command are supported (see !PXE.Implementation flags), multiple CPBs can be packed together. For each complete received frame, a receive buffer large enough to contain the entire unfragmented frame needs to be described in the CPB. Note that if a smaller than required buffer is provided, only a portion of the packet is received into the buffer, and the remainder of the packet is...
lost. Subsequent attempts to receive the same packet with a corrected (larger) buffer will be unsuccessful, because the packet will have been flushed from the queue.

```c
#pragma pack(1)
typedef struct s_pxe_cpb_receive {
    // Address of first byte of receive buffer. This is also the first byte of the frame header. This address must be a processor-based address for S/W UNDI and a device-based address for H/W UNDI.
    PXE_UINT64 BufferAddr;

    // Length of receive buffer. This must be large enough to hold the received frame (media header + data). If the length of smaller than the received frame, data will be lost.
    PXE_UINT32 BufferLen;

    // Reserved, must be set to zero.
    PXE_UINT32 reserved;
} PXE_CPB_RECEIVE;
#pragma pack()
```

### E.4.19.3 Waiting for the Command to Execute

E.4.19.3 Waiting for the Command to Execute

Monitor the upper two bits (14 & 15) in the `CDB.StatFlags` field. Until these bits change to report `PXE_STATFLAGS_COMMAND_COMPLETE` or `PXE_STATFLAGS_COMMAND_FAILED`, the command has not been executed by the UNDI.

<table>
<thead>
<tr>
<th>StatFlags</th>
<th>Reason</th>
</tr>
</thead>
<tbody>
<tr>
<td>COMMAND_COMPLETE</td>
<td>Command completed successfully. Frames received and DB is written.</td>
</tr>
<tr>
<td>COMMAND_QUEUED</td>
<td>Command has been queued.</td>
</tr>
<tr>
<td>INITIALIZE</td>
<td>Command has been not executed or queued.</td>
</tr>
</tbody>
</table>
E.4.19.4 Checking Command Execution Results

After command execution completes, either successfully or not, the CDB.StatCode field contains the result of the command execution.

<table>
<thead>
<tr>
<th>StatCode</th>
<th>Reason</th>
</tr>
</thead>
<tbody>
<tr>
<td>SUCCESS</td>
<td>Command completed successfully. Frames received and DB is written.</td>
</tr>
<tr>
<td>INVALID_CDB</td>
<td>One of the CDB fields was not set correctly.</td>
</tr>
<tr>
<td>INVALID_CPB</td>
<td>One of the CPB fields was not set correctly.</td>
</tr>
<tr>
<td>BUSY</td>
<td>UNDI is already processing commands. Try again later.</td>
</tr>
<tr>
<td>QUEUE_FULL</td>
<td>Command queue is full. Wait for queued commands to complete. Try again</td>
</tr>
<tr>
<td>NO_DATA</td>
<td>Receive buffers are empty.</td>
</tr>
<tr>
<td>NOT_STARTED</td>
<td>The UNDI is not started.</td>
</tr>
<tr>
<td>NOT_INITIALIZED</td>
<td>The UNDI is not initialized.</td>
</tr>
</tbody>
</table>

E.4.19.5 Using the DB

If multiple frames per command are supported (see `!PXE.Implementation` flags), multiple DBs can be packed together.

```c
#pragma pack(1)
typedef struct s_pxe_db_receive {
    // Source and destination MAC addresses from media header.
    PXE_MAC_ADDR SrcAddr;
    PXE_MAC_ADDR DestAddr;

    // Length of received frame. May be larger than receive buffer
    // size. The receive buffer will not be overwritten. This is
    // how to tell if data was lost because the receive buffer was
    // too small.
    PXE_UINT32 FrameLen;

    // Protocol type from media header.
    PXE_PROTOCOL Protocol;

    // Length of media header in received frame.
    PXE_UINT16 MediaHeaderLen;

    // Type of receive frame.
    PXE_FRAME_TYPE Type;

    // Reserved, must be zero.
    PXE_UINT8 reserved[7];
} PXE_DB_RECEIVE;
#pragma pack()
```
E.4.20 PXE 2.1 specification wire protocol clarifications

The Preboot Execution Environment (PXE) Version 2.1 specification was published in September 1999. Since then, this specification has not been maintained or updated for a new version. For adapting the IPv6 stack, the definition of an IPv6-based PXE process has been described in UEFI specification since version 2.2. Other clarifications for the IPv4-based PXE process defined in the PXE 2.1 specification are provided in this section.

E.4.20.1 Issue #1-time-outs

Where the PXE 2.1 specification reads:

DHCP Discover will be retried four times. The four timeouts are 4, 8, 16 and 32 seconds respectively.

If a DHCPOFFER is received without an Option #60 tag "PXEClient", DHCP Discover will be retried on the 4-and 8-second timeouts in an attempt to receive a PXE response.

Because of spanning tree algorithms in routers, the behavior should be as follows:

DHCP Discover will be retried four times. The four timeouts are 4, 8, 16 and 32 seconds respectively.

This process could be iterated three times.

If a DHCPOFFER is received without an Option #60 tag "PXEClient", DHCP Discover will be retried on the 4-and 8-second timeouts in an attempt to receive a PXE response.

E.4.20.2 Issue #2 - siaddr/option 54 precedence

Where the PXE 2.1 specification reads:

Boot server IP address (Read from the DHCP option 54 (server identifier), if not found, use the siaddr field.)

The behavior should be reversed, namely:

Ascertain the Boot server IP address from siaddr field. If not found, use the value in the DHCP option 54 (server identifier).

E.4.20.3 Issue #3 - PXE Vendor Options Existence

The PXE 2.1 specification is ambiguous about whether the following PXE Vendor Options need to be provided in DHCP messages. These options are marked as “Required” in Table 2-1 “PXE DHCP Options (Full List)”, but other parts of the specification state that these options may not be supplied in certain condition.

This section clarifies the existence of these PXE Vendor Options:

1. PXE_DISCOVERY_CONTROL (Tag 6)
   Where the PXE 2.1 specification reads:
   - Required, Note #3
   - If this tag is not supplied all bits assumed to be 0.
The behavior should be clarified as:
-This tag is not mandatory required. If not supplied, all bits are assumed to be 0.

2. **PXE_BOOT_SERVERS (Tag 8)**
Where the PXE 2.1 specification reads:
-Required for PXE client. Note #3
  -PXE_DISCOVERY_CONTROL (Tag 6), bit 2 = If set, only use/accept servers in PXE_BOOT_SERVERS.

The behavior should be clarified as:
-This tag is required only if bit 2 of PXE_DISCOVERY_CONTROL (Tag 6) is set.

3. **PXE_BOOT_MENU (Tag 9)**
Where the PXE 2.1 specification reads:
-Required, Note #4
  -Note #4: These options define the information, if any, displayed by the client during a network boot.

The behavior should be clarified as:
-This tag is required only if the PXE client wants to display boot menu information during a network boot.

4. **PXE_CREDENTIAL_TYPES (Tag 12)**
Where the PXE 2.1 specification reads:
-Required for security. Note #5
  -This option is required for security requests and acknowledges between the client and the server.

The behavior should be clarified as:
-This tag is not required if PXE client does not apply security requests.

5. **PXE_BOOT_ITEM (Tag 71)**
Where the PXE 2.1 specification reads:
-Required. Note #6
  -If this tag is missing, type 0 and layer 0 is assumed.

The behavior should be clarified as:
-This tag is not mandatory required. If not supplied, type 0 and layer 0 is assumed.

6. **Vendor Options (Tag 43)**
The PXE 2.1 specification is not clear whether this option is required.

The behavior should be clarified as:
-Vendor Options (Tag 43) is required only if encapsulated PXE options need be supplied.
Appendix F - Using the Simple Pointer Protocol

The Simple Pointer Protocol is intended to provide a simple mechanism for an application to interact with the user with some type of pointer device. To keep this interface simple, many of the custom controls that are typically present in an OS-present environment were left out. This includes the ability to adjust the double-click speed and the ability to adjust the pointer speed. Instead, the recommendations for how the Simple Pointer Protocol should be used are listed here.

X-Axis Movement:
If the Simple Pointer Protocol is being used to move a pointer or cursor around on an output display, the movement along the x-axis should move the pointer or cursor horizontally.

Y-Axis Movement:
If the Simple Pointer Protocol is being used to move a pointer or cursor around on an output display, the movement along the y-axis should move the pointer or cursor vertically.

Z-Axis Movement:
If the Simple Pointer Protocol is being used to move a pointer or cursor around on an output display, and the application that is using the Simple Pointer Protocol supports scrolling, then the movement along the z-axis should scroll the output display.

Double Click Speed:
If two clicks of the same button on a pointer occur in less than 0.5 seconds, then a double-click event has occurred. If a the same button is pressed with more than 0.5 seconds between clicks, then this is interpreted as two single-click events.

Pointer Speed:
The Simple Pointer Protocol returns the movement of the pointer device along an axis in counts. The Simple Pointer Protocol also contains a set of resolution fields that define the number of counts that will be received for each millimeter of movement of the pointer device along an axis. From these two values, the consumer of this protocol can determine the distance the pointer device has been moved in millimeters along an axis. For most applications, movement of a pointer device will result in the movement of a pointer on the screen. For each millimeter of motion by the pointer device in the x-axis, the pointer on the screen will be moved 2 percent of the screen width. For each millimeter of motion by the pointer device in the y-axis, the pointer on the screen will be moved 2 percent of the screen height.
Appendix G - Using the EFI Extended SCSI Pass Thru Protocol

This appendix describes how an EFI utility might gain access to the EFI SCSI Pass Thru interfaces. The basic concept is to use the EFI_BOOT_SERVICES.LocateHandle() boot service to retrieve the list of handles that support the EFI_EXT_SCSI_PASS_THRU_PROTOCOL. Each of these handles represents a different SCSI channel present in the system. Each of these handles can then be used to retrieve the EFI_EXT_SCSI_PASS_THRU_PROTOCOL interface with the EFI_BOOT_SERVICES.HandleProtocol() boot service. The EFI_EXT_SCSI_PASS_THRU_PROTOCOL interface provides the services required to access any of the SCSI devices attached to a SCSI channel. The services of the EFI_EXT_SCSI_PASS_THRU_PROTOCOL are then to loop through the Target IDs of all the SCSI devices on the SCSI channel.

```
#include "efi.h"
#include "efilib.h"

#include EFI_PROTOCOL_DEFINITION(ExtScsiPassThru)

EFI_GUID gEfiExtScsiPassThruProtocolGuid = 
  EFI_EXT_SCSI_PASS_THRU_PROTOCOL_GUID;

EFI_STATUS
UtilityEntryPoint(
  EFI_HANDLE     ImageHandle,
  EFI_SYSTEM_TABLE  SystemTable
)
{
  EFI_STATUS   Status;
  UINTN        NoHandles;
  EFI_HANDLE   *HandleBuffer;
  UINTN        Index;
  EFI_EXT_SCSI_PASS_THRU_PROTOCOL *ExtScsiPassThruProtocol;

  // Initialize EFI Library
  //
  InitializeLib (ImageHandle, SystemTable);

  // Get list of handles that support the
  // EFI_EXT_SCSI_PASS_THRU_PROTOCOL
  //
  NoHandles = 0;
  HandleBuffer = NULL;
  Status = LibLocateHandle(
    ByProtocol,
    &gEfiExtScsiPassThruProtocolGuid,
    NULL,
    &NoHandles,
    &HandleBuffer,
  );
```
if (EFI_ERROR(Status)) {
    BS->Exit(ImageHandle, EFI_SUCCESS, 0, NULL);
}

// Loop through all the handles that support // EFI_EXT_SCSI_PASS_THRU //
for (Index = 0; Index < NoHandles; Index++) {

    // Get the EFI_EXT_SCSI_PASS_THRU_PROTOCOL Interface // on each handle //
    BS->HandleProtocol(
        HandleBuffer[Index],
        &gEfiExtScsiPassThruProtocolGuid,
        (VOID **)&ExtScsiPassThruProtocol
    );

    if (!EFI_ERROR(Status)) {
        // Use the EFI_EXT_SCSI_PASS_THRU Interface to // perform tests //
        Status = DoScsiTests(ScsiPassThruProtocol);
    }
    return EFI_SUCCESS;
}

EFI_STATUS
DoScsiTests(
    EFI_EXT_SCSI_PASS_THRU_PROTOCOL *ExtScsiPassThruProtocol
)
{
    EFI_STATUS Status;
    UINT32 Target;
    UINT64 Lun;
    EFI_EXT_SCSI_PASS_THRU_SCSI_REQUEST_PACKET Packet;
    EFI_EVENT Event;

    // Get first Target ID and LUN on the SCSI channel //
    Target = 0xffffffff;
    Lun = 0;
    Status = ExtScsiPassThruProtocol-> GetNextTargetLun(
        ExtScsiPassThruProtocol,
        &Target,
        &Lun
    );
// Loop through all the SCSI devices on the SCSI channel
while (!EFI_ERROR (Status)) {
    // Blocking I/O example.
    // Fill in Packet before calling PassThru()
    Status = ExtScsiPassThruProtocol->PassThru(
        ExtScsiPassThruProtocol,
        Target,
        Lun,
        &Packet,
        NULL
    );

    // Non Blocking I/O
    // Fill in Packet and create Event before calling PassThru()
    Status = ExtScsiPassThruProtocol->PassThru(
        ExtScsiPassThruProtocol,
        Target,
        Lun,
        &Packet,
        &Event
    );

    // Get next Target ID and LUN on the SCSI channel
    Status = ExtScsiPassThruProtocol->GetNextTargetLun(
        ExtScsiPassThruProtocol,
        &Target,
        &Lun
    );
}
return EFI_SUCCESS;
Appendix H - Compression Source Code

/*++

Copyright (c) 2001–2002 Intel Corporation

Module Name:

Compress.c

Abstract:

Compression routine. The compression algorithm is a mixture of LZ77 and Huffman Coding. LZ77 transforms the source data into a sequence of Original Characters and Pointers to repeated strings. This sequence is further divided into Blocks and Huffman codings are applied to each Block.

Revision History:
--*/

#include <string.h>
#include <stdlib.h>
#include "eficommon.h"

//
// Macro Definitions
//

typedef INT16 NODE;
#define UINT8_MAX 0xff
#define UINT8_BIT 8
#define THRESHOLD 3
#define INIT_CRC 0
#define WNDBIT 13
#define WNDDBIT (1U << WNDBIT)
#define MAXMATCH 256
#define PERC_FLAG 0x8000U
#define CODE_BIT 16
#define NIL 0
#define NC (UINT8_MAX + MAXMATCH + 2 - THRESHOLD)
#define CBIT 9
#define NP (WNDBIT + 1)

#define MAX_HASH_VAL (3 * WNDDBIT + (WNDDBIT / 512 + 1) * UINT8_MAX)
#define HASH(p, c) (((p) + ((c) << (WNDBIT - 9))) + WNDDBIT * 2)
#define CRCPOLY 0xA001
#define UPDATE_CRC(c) mCrc = mCrcTable[(mCrc ^ (c)) & 0xFF] ^ (mCrc >> UINT8_BIT)

//
// C: the Char&Len Set; P: the Position Set; T: the exTra Set
//
#define NC (UINT8_MAX + MAXMATCH + 2 - THRESHOLD)
#define CBIT 9
#define NP (WNDBIT + 1)
```
#define PBIT 4
#define NT (CODE_BIT + 3)
#define TBIT 5
#if NT > NP
#define NPT NT
#else
#define NPT NP
#endif

// Function Prototypes
//
STATIC VOID PutDword(
    IN UINT32 Data
);

STATIC EFI_STATUS AllocateMemory (
);

STATIC VOID FreeMemory ( 
);

STATIC VOID InitSlide ( 
);

STATIC NODE Child ( 
    IN NODE q,
    IN UINT8 c
);

STATIC VOID MakeChild ( 
    IN NODE q,
    IN UINT8 c,
    IN NODE r
);

STATIC VOID Split ( 
    IN NODE Old
);
```
STATIC VOID InsertNode ( );

STATIC VOID DeleteNode ( );

STATIC VOID getNextMatch ( );

STATIC EFI_STATUS Encode ( );

STATIC VOID CountTFreq ( );

STATIC VOID WritePTLen ( IN INT32 n, IN INT32 nbit, IN INT32 Special );

STATIC VOID WriteCLen ( );

STATIC VOID EncodeC ( IN INT32 c );

STATIC VOID EncodeP ( IN UINT32 p );

STATIC VOID SendBlock ( );
STATIC VOID
Output (IN UINT32 c, IN UINT32 p);

STATIC VOID
HufEncodeStart ();

STATIC VOID
HufEncodeEnd ();

STATIC VOID
MakeCrcTable ();

STATIC VOID
PutBits (IN INT32 n, IN UINT32 x);

STATIC INT32
FreadCrc (OUT UINT8 *p, IN INT32 n);

STATIC VOID
InitPutBits ();

STATIC VOID
CountLen (IN INT32 i);

STATIC VOID
MakeLen (IN INT32 Root);

STATIC VOID
DownHeap (  
    IN INT32 i  
);  

STATIC  
VOID  
MakeCode (  
    IN INT32 n,  
    IN UINT8 Len[],  
    OUT UINT16 Code[]  
);  

STATIC  
INT32  
MakeTree (  
    IN INT32 NParm,  
    IN UINT16 FreqParm[],  
    OUT UINT8 LenParm[],  
    OUT UINT16 CodeParm[]  
);  

// // Global Variables  
//  
STATIC UINT8 *mSrc, *mDst, *mSrcUpperLimit, *mDstUpperLimit;  
STATIC UINT8 *mLevel, *mText, *mChildCount, *mBuf, mLen[NC], mPTLen[NPT],  
    *mLen;  
STATIC INT16 mHeap[NC + 1];  
STATIC INT32 mRemainder, mMatchLen, mBitCount, mHeapSize, mN;  
STATIC UINT32 mBufSiz = 0, mOutputPos, mOutputMask, mSubBitBuf, mCrc;  
STATIC UINT32 mCompSize, mOrigSize;  
STATIC UINT16 *mFreq, *mSortPtr, mLenCnt[17], mLft[2 * NC - 1], mRght[2 * NC - 1],  
    mCrcTable[UINT8_MAX + 1], mCFreq[2 * NC - 1], mCTable[4096], mCCode[NC],  
    mPFreq[2 * NP - 1], mPTCode[NPT], mTFreq[2 * NT - 1];  
STATIC NODE mPos, mMatchPos, mAvail, *mPosition, *mParent, *mPrev, *mNext = NULL;  

// // functions  
//  
EFI_STATUS  
Compress (  
    IN UINT8 *SrcBuffer,  
    IN UINT32 SrcSize,  
    IN UINT8 *DstBuffer,  
    IN OUT UINT32 *DstSize  
)
/**

Routine Description:

The main compression routine.

Arguments:

SrcBuffer - The buffer storing the source data
SrcSize   - The size of the source data
DstBuffer - The buffer to store the compressed data
DstSize   - On input, the size of DstBuffer; On output, the size of the actual compressed data.

Returns:

EFI_BUFFER_TOO_SMALL - The DstBuffer is too small. In this case, DstSize contains the size needed.
EFI_SUCCESS      - Compression is successful.

--*/
{
  EFI_STATUS Status = EFI_SUCCESS;

  // Initializations
  //
  mBufSiz = 0;
  mBuf = NULL;
  mText    = NULL;
  mLevel   = NULL;
  mChildCount = NULL;
  mPosition  = NULL;
  mParent   = NULL;
  mPrev    = NULL;
  mNext    = NULL;

  mSrc = SrcBuffer;
  mSrcUpperLimit = mSrc + SrcSize;
  mDst = DstBuffer;
  mDstUpperLimit = mDst + *DstSize;

  PutDword(0L);
  PutDword(0L);

  MakeCrcTable();

  mOrigSize = mCompSize = 0;
  mCrc = INIT_CRC;

  //
  // Compress it
  //
Status = Encode();
if (EFI_ERROR (Status)) {
    return EFI_OUT_OF_RESOURCES;
}

// Null terminate the compressed data
//
if (mDst < mDstUpperLimit) {
    *mDst++ = 0;
}

// Fill in compressed size and original size
//
mDst = DstBuffer;
PutDword(mCompSize+1);
PutDword(mOrigSize);

// Return
//
if (mCompSize + 1 + 8 > *DstSize) {
    *DstSize = mCompSize + 1 + 8;
    return EFI_BUFFER_TOO_SMALL;
} else {
    *DstSize = mCompSize + 1 + 8;
    return EFI_SUCCESS;
}

STATIC
VOID
PutDword(
    IN UINT32 Data
)
/*++

Routine Description:

    Put a dword to output stream

Arguments:

    Data - the dword to put

Returns: (VOID)

--*/
{
    if (mDst < mDstUpperLimit) {
        *mDst++ = (UINT8)(((UINT8)(Data)) & 0xff);
    }
if (mDst < mDstUpperLimit) {
    *mDst++ = (UINT8)(((UINT8)(Data >> 0x08)) & 0xff);
}
if (mDst < mDstUpperLimit) {
    *mDst++ = (UINT8)(((UINT8)(Data >> 0x10)) & 0xff);
}
if (mDst < mDstUpperLimit) {
    *mDst++ = (UINT8)(((UINT8)(Data >> 0x18)) & 0xff);
}
}

STATIC
EFI_STATUS
AllocateMemory ()
/*++
Routine Description:
Allocate memory spaces for data structures used in compression process
Arguments: (VOID)
Returns:
EFI_SUCCESS      - Memory is allocated successfully
EFI_OUT_OF_RESOURCES - Allocation fails
--*/
{
    UINT32 i;
    mText = malloc (WNDSIZ * 2 + MAXMATCH);
    for (i = 0; i < WNDSIZ * 2 + MAXMATCH; i++) {
        mText[i] = 0;
    }
    mLevel   = malloc ((WNDSIZ + UINT8_MAX + 1) * sizeof(*mLevel));
    mChildCount = malloc ((WNDSIZ + UINT8_MAX + 1) * sizeof(*mChildCount));
    mPosition  = malloc ((WNDSIZ + UINT8_MAX + 1) * sizeof(*mPosition));
    mParent   = malloc (WNDSIZ * 2 * sizeof(*mParent));
    mPrev    = malloc (WNDSIZ * 2 * sizeof(*mPrev));
    mNext    = malloc ((MAX_HASH_VAL + 1) * sizeof(*mNext));
    mBufSiz = 16 * 1024U;
    while ((mBuf = malloc(mBufSiz)) == NULL) {
        mBufSiz = (mBufSiz / 10U) * 9U;
        if (mBufSiz < 4 * 1024U) {
            return EFI_OUT_OF_RESOURCES;
        }
    }
    mBuf[0] = 0;
return EFI_SUCCESS;
}

VOID
FreeMemory ()
/**++
Routine Description:
Called when compression is completed to free memory previously allocated.
Arguments: (VOID)
Returns: (VOID)
--*/
{
  if (mText) {
    free (mText);
  }
  if (mLevel) {
    free (mLevel);
  }
  if (mChildCount) {
    free (mChildCount);
  }
  if (mPosition) {
    free (mPosition);
  }
  if (mParent) {
    free (mParent);
  }
  if (mPrev) {
    free (mPrev);
  }
  if (mNext) {
    free (mNext);
  }
  if (mBuf) {
    free (mBuf);
  }
  return;
}

STATIC
VOID
InitSlide ()
/*++

Routine Description:

Initialize String Info Log data structures

Arguments: (VOID)

Returns: (VOID)
--*/
{
    NODE i;
    for (i = WNDSIZ; i <= WNDSIZ + UINT8_MAX; i++) {
        mLevel[i] = 1;
        mPosition[i] = NIL; /* sentinel */
    }
    for (i = WNDSIZ; i < WNDSIZ * 2; i++) {
        mParent[i] = NIL;
    }
    mAvail = 1;
    for (i = 1; i < WNDSIZ - 1; i++) {
        mNext[i] = (NODE)(i + 1);
    }
    mNext[WNDSIZ - 1] = NIL;
    for (i = WNDSIZ * 2; i <= MAX_HASH_VAL; i++) {
        mNext[i] = NIL;
    }
}

STATIC
NODE
Child (   
    IN NODE q,
    IN UINT8 c
    )
/*++

Routine Description:

Find child node given the parent node and the edge character

Arguments:

    q    - the parent node
    c    - the edge character

Returns:

    The child node (NIL if not found)
NODE r;

r = mNext[HASH(q, c)];
mParent[NIL] = q; /* sentinel */
while (mParent[r] != q) {
    r = mNext[r];
}

return r;
}

STATIC VOID MakeChild (IN NODE q, IN UINT8 c, IN NODE r)
/*++
Routine Description:
Create a new child for a given parent node.
Arguments:
    q    - the parent node
    c    - the edge character
    r    - the child node
Returns: (VOID)
/*++

{ NODE h, t;

    h = (NODE)HASH(q, c);
    t = mNext[h];
    mNext[h] = r;
    mNext[r] = t;
    mPrev[t] = r;
    mPrev[r] = h;
    mParent[r] = q;
    mChildCount[q]++;
}

STATIC VOID Split (NODE Old)
/*++
Routine Description:

Split a node.

Arguments:

Old   - the node to split

Returns: (VOID)

--*/
{
    NODE New, t;

    New = mAvail;
    mAvail = mNext[New];
    mChildCount[New] = 0;
    t = mPrev[Old];
    mPrev[New] = t;
    mNext[t] = New;
    t = mNext[Old];
    mNext[New] = t;
    mPrev[t] = New;
    mParent[New] = mParent[Old];
    mLevel[New] = (UINT8)mMatchLen;
    mPosition[New] = mPos;
    MakeChild(New, mText[mMatchPos + mMatchLen], Old);
    MakeChild(New, mText[mPos + mMatchLen], mPos);
}

STATIC
VOID
InsertNode ()
/*++

Routine Description:

Insert string info for current position into the String Info Log

Arguments: (VOID)

Returns: (VOID)

--*/
{
    NODE q, r, j, t;
    UINT8 c, *t1, *t2;

    if (mMatchLen >= 4) {
        //
        // We have just got a long match, the target tree
        // can be located by MatchPos + 1. Traverse the tree
        // from bottom up to get to a proper starting point.
        // The usage of PERC_FLAG ensures proper node deletion
// in DeleteNode() later.

mMatchLen--;
r = (INT16)((mMatchPos + 1) | WNDSIZ);
while ((q = mParent[r]) == NIL) {
    r = mNext[r];
} while (mLevel[q] >= mMatchLen) {
    r = q; q = mParent[q];
} t = q;
while (mPosition[t] < 0) {
    mPosition[t] = mPos;
    t = mParent[t];
} if (t < WNDSIZ) {
    mPosition[t] = (NODE)(mPos | PERC_FLAG);
} else {

    // Locate the target tree
    //

    q = (INT16)(mText[mPos] + WNDSIZ); c = mText[mPos + 1];
    if ((r = Child(q, c)) == NIL) {
        MakeChild(q, c, mPos);
        mMatchLen = 1;
        return;
    } mMatchLen = 2;
}

    // Traverse down the tree to find a match.
    // Update Position value along the route.
    // Node split or creation is involved.
    //

    for (; ; ) {
        if (r >= WNDSIZ) {
            j = MAXMATCH;
            mMatchPos = r;
        } else {
            j = mLevel[r];
            mMatchPos = (NODE)(mPosition[r] & ~PERC_FLAG);
        } if (mMatchPos >= mPos) {
            mMatchPos -= WNDSIZ;
        } t1 = &mText[mPos + mMatchLen];
        t2 = &mText[mMatchPos + mMatchLen];
        while (mMatchLen < j) {
if (*t1 != *t2) {
    Split(r);
    return;
}  
++mMatchLen;
++t1;
++t2;
}  
if (mMatchLen >= MAXMATCH) {
    break;
} 
++mPosition[r] = mPos;
++q = r;
if ((r = Child(q, *t1)) == NIL) {
    MakeChild(q, *t1, mPos);
    return;
} 
++mMatchLen;
} 
++t = mPrev[r];
++mPrev[mPos] = t;
++mNext[t] = mPos;
++t = mNext[r];
++mNext[mPos] = t;
++mPrev[t] = mPos;
++mParent[mPos] = q;
++mParent[r] = NIL;

// Special usage of 'next'
// mNext[r] = mPos;

} 

STATIC
VOID
DeleteNode ()
/**
Routine Description:
  Delete outdated string info. (The Usage of PERC_FLAG ensures a clean deletion)
Arguments: (VOID)
Returns: (VOID)
--*/
{
    NODE q, r, s, t, u;
    if (mParent[mPos] == NIL) {
        return;
    }
r = mPrev[mPos];
s = mNext[mPos];
mNext[r] = s;
mPrev[s] = r;
r = mParent[mPos];
mParent[mPos] = NIL;
if (r >= WNDSIZ || --mChildCount[r] > 1) {
    return;
}
t = (NODE)(mPosition[r] & ~PERC_FLAG);
if (t >= mPos) {
    t -= WNDSIZ;
}
s = t;
q = mParent[r];
while ((u = mPosition[q]) & PERC_FLAG) {
    u &= ~PERC_FLAG;
    if (u >= mPos) {
        u -= WNDSIZ;
    }
    if (u > s) {
        s = u;
    }
    mPosition[q] = (INT16)(s | WNDSIZ);
    q = mParent[q];
}
if (q < WNDSIZ) {
    if (u >= mPos) {
        u -= WNDSIZ;
    }
    if (u > s) {
        s = u;
    }
    mPosition[q] = (INT16)(s | WNDSIZ | PERC_FLAG);
}
s = Child(r, mText[t + mLevel[r]]);
t = mPrev[s];
u = mNext[s];
mNext[t] = u;
mPrev[u] = t;
t = mPrev[r];
mNext[t] = s;
mPrev[s] = t;
t = mNext[r];
mPrev[t] = s;
mNext[s] = t;
mParent[s] = mParent[r];
mParent[r] = NIL;
mNext[r] = mAvail;
mAvail = r;
}
VOID
GetNextMatch ()
/**++

Routine Description:

Advance the current position (read in new data if needed).
Delete outdated string info. Find a match string for current position.

Arguments: (VOID)

Returns: (VOID)

--*/
{
    INT32 n;

    mRemainder--;       
    if (++mPos == WNDSIZ * 2) {
        memmove(&mText[0], &mText[WNDSIZ], WNDSIZ + MAXMATCH);
        n = FreadCrc(&mText[WNDSIZ + MAXMATCH], WNDSIZ);
        mRemainder += n;
        mPos = WNDSIZ;
    }  
    DeleteNode();
    InsertNode();
}

STATIC
EFI_STATUS
Encode ()
/**++

Routine Description:

The main controlling routine for compression process.

Arguments: (VOID)

Returns:

    EFI_SUCCESS      - The compression is successful
    EFI_OUT_OF_RESOURCES - Not enough memory for compression process

--*/
{
    EFI_STATUS Status;
    INT32    LastMatchLen;
    NODE     LastMatchPos;

    Status = AllocateMemory();
    if (EFI_ERROR(Status)) {
        FreeMemory();
        return Status;
    }
InitSlide();

HufEncodeStart();

mRemainder = FreadCrc(&mText[WNDSIZ], WNDSIZ + MAXMATCH);

mMatchLen = 0;
mPos = WNDSIZ;
InsertNode();
if (mMatchLen > mRemainder) {
    mMatchLen = mRemainder;
}
while (mRemainder > 0) {
    LastMatchLen = mMatchLen;
    LastMatchPos = mMatchPos;
    GetNextMatch();
    if (mMatchLen > mRemainder) {
        mMatchLen = mRemainder;
    }
    if (mMatchLen > LastMatchLen || LastMatchLen < THRESHOLD) {
        //
        // Not enough benefits are gained by outputting a pointer,
        // so just output the original character
        //
        Output(mText[mPos - 1], 0);
    } else {
        //
        // Outputting a pointer is beneficial enough, do it.
        //
        Output(LastMatchLen + (UINT8_MAX + 1 - THRESHOLD),
               (mPos - LastMatchPos - 2) & (WNDSIZ - 1));
        while (--LastMatchLen > 0) {
            GetNextMatch();
        }
        if (mMatchLen > mRemainder) {
            mMatchLen = mRemainder;
        }
    }
}

HufEncodeEnd();
FreeMemory();
return EFI_SUCCESS;

STATIC
VOID
CountTFreq ()
/++;
Routine Description:

Count the frequencies for the Extra Set

Arguments: (VOID)

Returns: (VOID)

```c
{         
    INT32 i, k, n, Count;
    for (i = 0; i < NT; i++) {
        mTFreq[i] = 0;
    }              
    n = NC;
    while (n > 0 && mCLen[n - 1] == 0) {  
        n--;
    }            
    i = 0;
    while (i < n) {  
        k = mCLen[i++];
        if (k == 0) {  
            Count = 1;
            while (i < n && mCLen[i] == 0) {  
                i++;
                Count++;
            }                           
            if (Count <= 2) {  
                mTFreq[0] = (UINT16)(mTFreq[0] + Count);
            } else if (Count <= 18) {  
                mTFreq[1]++;
            } else if (Count == 19) {  
                mTFreq[0]++;
                mTFreq[1]++;
            } else {  
                mTFreq[2]++;
            }                     
        } else {  
            mTFreq[k + 2]++;
        }    }
}

STATIC
VOID
WritePTLen(
    IN INT32 n,
    IN INT32 nbit,
    IN INT32 Special
) /*++

Routine Description:
Outputs the code length array for the Extra Set or the Position Set.

Arguments:

n    - the number of symbols
nbit - the number of bits needed to represent 'n'
Special - the special symbol that needs to be take care of

Returns: (VOID)

--*/
{
    INT32 i, k;
    while (n > 0 && mPTLen[n - 1] == 0) {
        n--;
    }
    PutBits(nbit, n);
    i = 0;
    while (i < n) {
        k = mPTLen[i++];
        if (k <= 6) {
            PutBits(3, k);
        } else {
            PutBits(k - 3, (1U << (k - 3)) - 2);
        }
        if (i == Special) {
            while (i < 6 && mPTLen[i] == 0) {
                i++;
            }
            PutBits(2, (i - 3) & 3);
        }
    }
}

STATIC
VOID
WriteCLen ()
/*++

Routine Description:

Outputs the code length array for Char&Length Set

Arguments: (VOID)

Returns: (VOID)

--*/
{
    INT32 i, k, n, Count;
    n = NC;
    while (n > 0 && mCLen[n - 1] == 0) {

n--; } 
PutBits(CBIT, n); 
i = 0; 
while (i < n) {
  k = mCLen[i++]; 
  if (k == 0) {
    Count = 1; 
    while (i < n && mCLen[i] == 0) {
      i++; 
      Count++; 
    } 
    if (Count <= 2) {
      for (k = 0; k < Count; k++) {
        PutBits(mPTLen[0], mPTCode[0]);
      } 
    } else if (Count <= 18) {
      PutBits(mPTLen[1], mPTCode[1]); 
      PutBits(4, Count - 3); 
    } else if (Count == 19) {
      PutBits(mPTLen[0], mPTCode[0]);
      PutBits(mPTLen[1], mPTCode[1]);
      PutBits(4, 15); 
    } else {
      PutBits(mPTLen[2], mPTCode[2]);
      PutBits(CBIT, Count - 20); 
    } 
  } 
} 

STATIC VOID EncodeC ( 
  IN INT32 c 
) 
{ 
  PutBits(mCLen[c], mCCode[c]); 
}

STATIC VOID EncodeP ( 
  IN UINT32 p 
) 
{ 
  UINT32 c, q; 

  c = 0; 
  q = p; 
  while (q) { 
    q >>= 1; 
    c++; 
  }
} 
PutBits(mPTLen[c], mPTCode[c]);
if (c > 1) {
  PutBits(c - 1, p & (0xFFFFU >> (17 - c)));
}
}

STATIC
VOID
SendBlock ()
/*++

Routine Description:

Huffman code the block and output it.

Argument: (VOID)

Returns: (VOID)
--*/
{
  UINT32 i, k, Flags, Root, Pos, Size;
  Flags = 0;

  Root = MakeTree(NC, mCFreq, mCLen, mCCode);
  Size = mCFreq[Root];
  PutBits(16, Size);
  if (Root >= NC) {
    CountTFreq();
    Root = MakeTree(NT, mTFreq, mPTLen, mPTCode);
    if (Root >= NT) {
      WritePTLen(NT, TBIT, 3);
    } else {
      PutBits(TBIT, 0);
      PutBits(TBIT, Root);
    }
  } else {
    PutBits(TBIT, 0);
    PutBits(TBIT, 0);
    PutBits(CBIT, 0);
    PutBits(CBIT, Root);
  }
  WriteCLen();
} else {
  PutBits(TBIT, 0);
  PutBits(TBIT, 0);
  PutBits(CBIT, 0);
  PutBits(CBIT, Root);
}

Root = MakeTree(NP, mPFreq, mPTLen, mPCode);
if (Root >= NP) {
  WritePTLen(NP, PBIT, -1);
} else {
  PutBits(PBIT, 0);
  PutBits(PBIT, Root);
}

Pos = 0;
for (i = 0; i < Size; i++) {
  if (i % UINT8_BIT == 0) {
    Flags = mBuf[Pos++];
  }
  PutBits(mPTLen[i], mPTCode[i]);
}
else {
    Flags <<= 1;
}
if (Flags & (1U << (UINT8_BIT - 1))) {
    EncodeC(mBuf[Pos++]);
    k = mBuf[Pos++] << UINT8_BIT;
    k += mBuf[Pos++];
    EncodeP(k);
} else {
    EncodeC(mBuf[Pos++]);
}
for (i = 0; i < NC; i++) {
    mCFreq[i] = 0;
}
for (i = 0; i < NP; i++) {
    mPFreq[i] = 0;
}

STATIC
VOID
Output (IN UINT32 c, IN UINT32 p)
/*++
Routine Description:
Outputs an Original Character or a Pointer
Arguments:
c - The original character or the 'String Length' element of a Pointer
p - The 'Position' field of a Pointer
Returns: (VOID)
--*/
{
    STATIC UINT32 CPos;
    if ((mOutputMask >>= 1) == 0) {
        mOutputMask = 1U << (UINT8_BIT - 1);
        if (mOutputPos >= mBufSiz - 3 * UINT8_BIT) {
            SendBlock();
            mOutputPos = 0;
        }
        CPos = mOutputPos++;
        mBuf[CPos] = 0;
    }
    mBuf[mOutputPos++] = (UINT8) c;
    mCFreq[c]++;
}
if (c >= (1U << UINT8_BIT)) {  
  mBuf[CPos] |= mOutputMask;  
  mBuf[mOutputPos++] = (UINT8)(p >> UINT8_BIT);  
  mBuf[mOutputPos++] = (UINT8) p;  
  c = 0;  
  while (p) {  
    p >>= 1;  
    c++;  
  }  
  mPFreq[c]++;  
}

STATIC  
VOID  
HufEncodeStart ()  
{  
  INT32 i;  
  for (i = 0; i < NC; i++) {  
    mCFreq[i] = 0;  
  }  
  for (i = 0; i < NP; i++) {  
    mPFreq[i] = 0;  
  }  
  mOutputPos = mOutputMask = 0;  
  InitPutBits();  
  return;  
}

STATIC  
VOID  
HufEncodeEnd ()  
{  
  SendBlock();  
  //  
  // Flush remaining bits  
  //  
  // PutBits(UINT8_BIT - 1, 0);  
  return;  
}

STATIC  
VOID  
MakeCrcTable ()  
{  
  UINT32 i, j, r;  
  for (i = 0; i <= UINT8_MAX; i++) {  
    r = i;  
    for (j = 0; j < UINT8_BIT; j++) {  
      if (r & 1) {  
        r = (r >> 1) ^ 0x1b;  
      } else {  
        r = r >> 1;  
      }  
    }  
    mCrcTable[i] = r;  
  }  
}
r = (r >> 1) ^ CRCPOLY;
} else {
    r >>= 1;
}
}
mCrcTable[i] = (UINT16)r;
}

STATIC
VOID
PutBits (  
    IN INT32 n,
    IN UINT32 x
)  
/

Routine Description:

Outputs rightmost n bits of x

Arguments:

n  - the rightmost n bits of the data is used
x  - the data

Returns: (VOID)

--*/
{
    UINT8 Temp;
    if (n < mBitCount) {
        mSubBitBuf |= x << (mBitCount -= n);
    } else {
        Temp = (UINT8)(mSubBitBuf | (x >> (n -= mBitCount)));
        if (mDst < mDstUpperLimit) {
            *mDst++ = Temp;
        }
        mCompSize++;
        if (n < UINT8_BIT) {
            mSubBitBuf = x << (mBitCount = UINT8_BIT - n);
        } else {
            Temp = (UINT8)(x >> (n - UINT8_BIT));
            if (mDst < mDstUpperLimit) {
                *mDst++ = Temp;
            }
            mCompSize++;
            mSubBitBuf = x << (mBitCount = 2 * UINT8_BIT - n);
        }
    }
}


```c
STATIC
INT32
FreadCrc (
    OUT UINT8 *p,
    IN INT32 n
)
/**++

Routine Description:

Read in source data

Arguments:

    p    - the buffer to hold the data
    n    - number of bytes to read

Returns:

    number of bytes actually read

--*/
{
    INT32 i;

    for (i = 0; mSrc < mSrcUpperLimit && i < n; i++) {
        *p++ = *mSrc++;
    }
    n = i;

    p -= n;
    mOrigSize += n;
    while (--i >= 0) {
        UPDATE_CRC(*p++);
    }
    return n;
}

STATIC
VOID
InitPutBits ()
{
    mBitCount = UINT8_BIT;
    mSubBitBuf = 0;
}

STATIC
VOID
CountLen ( IN INT32 i )
/++
```
Routine Description:

Count the number of each code length for a Huffman tree.

Arguments:

i  - the top node

Returns: (VOID)

```
{  
  STATIC INT32 Depth = 0;

  if (i < mN) {
    mLLenCnt[(Depth < 16) ? Depth : 16]++;
  } else {
    Depth++;
    CountLen(mLeft [i]);
    CountLen(mRight[i]);
    Depth--;
  }
}

STATIC VOID
MakeLen (  
    IN INT32 Root
  )
/*++

Routine Description:

Create code length array for a Huffman tree

Arguments:

Root  - the root of the tree

--*/
{
  INT32 i, k;
  UINT32 Cum;

  for (i = 0; i <= 16; i++) {
    mLLenCnt[i] = 0;
  }
  CountLen(Root);

  // Adjust the length count array so that  
  // no code will be generated longer than the designated length  
  //
```
Cum = 0;
for (i = 16; i > 0; i--) {
    Cum += mLenCnt[i] << (16 - i);
}
while (Cum != (1U << 16)) {
    mLenCnt[16]--;
    for (i = 15; i > 0; i--) {
        if (mLenCnt[i] != 0) {
            mLenCnt[i]--;
            mLenCnt[i+1] += 2;
            break;
        }
    }
    Cum--;
}
for (i = 16; i > 0; i--) {
    k = mLenCnt[i];
    while (--k >= 0) {
        mLen[mSortPtr++] = (UINT8)i;
    }
}
}

STATIC
VOID
DownHeap (IN INT32 i)
{
    INT32 j, k;
    // priority queue: send i-th entry down heap
    k = mHeap[i];
    while ((j = 2 * i) <= mHeapSize) {
        if (j < mHeapSize && mFreq[mHeap[j]] > mFreq[mHeap[j + 1]]) {
            j++;
        }
        if (mFreq[k] <= mFreq[mHeap[j]]) {
            break;
        }
        mHeap[i] = mHeap[j];
        i = j;
    }
    mHeap[i] = (INT16)k;
}

STATIC
VOID
MakeCode (IN INT32 n, IN UINT8 Len[], OUT UINT16 Code[]
Routine Description:

Assign code to each symbol based on the code length array

Arguments:

- \( n \) - number of symbols
- \( \text{Len} \) - the code length array
- \( \text{Code} \) - stores codes for each symbol

Returns: (VOID)

```c
{  
  INT32  i;
  UINT16  Start[18];

  Start[1] = 0;
  for (i = 1; i <= 16; i++) {
    Start[i + 1] = (UINT16)((Start[i] + mLenCnt[i]) << 1);
  }
  for (i = 0; i < n; i++) {
    Code[i] = Start[Len[i]]++;
  }
}
```

STATIC
INT32
MakeTree (
  IN INT32  NParm,
  IN UINT16 FreqParm[],
  OUT UINT8  LenParm[],
  OUT UINT16 CodeParm[]
)

Routine Description:

Generates Huffman codes given a frequency distribution of symbols

Arguments:

- \( NParm \) - number of symbols
- \( \text{FreqParm} \) - frequency of each symbol
- \( \text{LenParm} \) - code length for each symbol
- \( \text{CodeParm} \) - code for each symbol

Returns:

Root of the Huffman tree.

```c
//**+
```
```c
{  
  INT32 i, j, k, Avail;

  // make tree, calculate len[], return root
  //
  mN = NParm;
  mFreq = FreqParm;
  mLen = LenParm;
  Avail = mN;
  mHeapSize = 0;
  mHeap[1] = 0;
  for (i = 0; i < mN; i++) {
    mLen[i] = 0;
    if (mFreq[i]) {
      mHeap[++mHeapSize] = (INT16)i;
    }
  }
  if (mHeapSize < 2) {
    CodeParm[mHeap[1]] = 0;
    return mHeap[1];
  }
  for (i = mHeapSize / 2; i >= 1; i--) {
    // make priority queue
    //
    DownHeap(i);
  }
  mSortPtr = CodeParm;
  do {
    i = mHeap[1];
    if (i < mN) {
      *mSortPtr++ = (UINT16)i;
    }
    mHeap[1] = mHeap[mHeapSize--];
    DownHeap(1);
    j = mHeap[1];
    if (j < mN) {
      *mSortPtr++ = (UINT16)j;
    }
    k = Avail++;
    mFreq[k] = (UINT16)(mFreq[i] + mFreq[j]);
    mHeap[1] = (INT16)k;
    DownHeap(1);
    mLeft[k] = (UINT16)i;
    mRight[k] = (UINT16)j;
  } while (mHeapSize > 1);
  mSortPtr = CodeParm;
  MakeLen(k);
  MakeCode(NParm, LenParm, CodeParm);
  //
```
// return root
//
return k;
}
/*++

Copyright (c) 2001–2002 Intel Corporation

Module Name:

Decompress.c
Abstract:
Decompressor.
--*/

#include "EfiCommon.h"

#define BITBUFSIZ 16
#define WNDBIT 13
#define WNDSIZE (1U << WNDBIT)
#define MAXMATCH 256
#define THRESHOLD 3
#define CODE_BIT 16
#define UINT8_MAX 0xff
#define BAD_TABLE -1

// C: Char&Len Set; P: Position Set; T: exTra Set
//
#define NC (0xff + MAXMATCH + 2 - THRESHOLD)
#define CBIT 9
#define NP (WNDBIT + 1)
#define NT (CODE_BIT + 3)
#define PBIT 4
#define TBIT 5
#if NT > NP
#define NPT NT
#else
#define NPT NP
#endif

typedef struct {
    UINT8 *mSrcBase;  //Starting address of compressed data
    UINT8 *mDstBase;  //Starting address of decompressed data
    UINT16 mBytesRemain;
    UINT16 mBitCount;
    UINT16 mBitBuf;
    UINT16 mSubBitBuf;
}
UINT16 mBufSiz;
UINT16 mBlockSize;
UINT32 mDataIdx;
UINT32 mCompSize;
UINT32 mOrigSize;
UINT32 mOutBuf;
UINT32 mInBuf;

UINT16 mBadTableFlag;

UINT8 mBuffer[WNDSIZ];
UINT16 mLef{2 * NC - 1];
UINT16 mRight[2 * NC - l];
UINT32 mBuf;
UINT8 mCLen[NC];
UINT8 mPTLen[NPT];
UINT16 mCTable[4096];
UINT16 mPTTTable[256];
} SCRATCH_DATA;

// Function Prototypes
//

STATIC VOID FillBuf (
    IN SCRATCH_DATA *Sd,
    IN UINT16 NumOfBits
);

STATIC VOID Decode ( 
    SCRATCH_DATA *Sd,
    UINT16 NumOfBytes
);

// Functions
//

EFI_STATUS EFIAP In GetInfo ( 
    IN EFI DECOMPRESS_PROTOCOL *This,
    IN VOID *Source,
    IN UINT32 SrcSize,
    OUT UINT32 *DstSize,
    OUT UINT32 *ScratchSize
) /*++
Routine Description:

The implementation of EFI_DECOMPRESS_PROTOCOL.GetInfo().

Arguments:

This - Protocol instance pointer.
Source - The source buffer containing the compressed data.
SrcSize - The size of source buffer
DstSize - The size of destination buffer.
ScratchSize - The size of scratch buffer.

Returns:

EFI_SUCCESS - The size of destination buffer and the size of scratch buffer are successful retrieved.
EFI_INVALID_PARAMETER - The source data is corrupted.

```c
{UINT8 *Src;

*ScratchSize = sizeof (SCRATCH_DATA);

Src = Source;
if (SrcSize < 8) {
    return EFI_INVALID_PARAMETER;
}

return EFI_SUCCESS;
}
```

```c
EFI_STATUS
EFI_API
Decompress ( 
    IN EFI_DECOMPRESS_PROTOCOL *This,
    IN VOID *Source,
    IN UINT32 SrcSize,
    IN OUT VOID *Destination,
    IN UINT32 DstSize,
    IN OUT VOID *Scratch,
    IN UINT32 ScratchSize
)
/*++
Routine Description:

The implementation of EFI_DECOMPRESS_PROTOCOL.Decompress().

Arguments:

This - The protocol instance.
Source - The source buffer containing the compressed data.
```
SrcSize   - The size of the source buffer
Destination - The destination buffer to store the decompressed data
DstSize   - The size of the destination buffer.
Scratch   - The buffer used internally by the decompress routine. This buffer is needed to store intermediate data.
ScratchSize - The size of scratch buffer.

Returns:

EFI_SUCCESS      - Decompression is successful
EFI_INVALID_PARAMETER - The source data is corrupted
--*/
{
    UINT32    Index;
    UINT16    Count;
    UINT32    CompSize;
    UINT32    OrigSize;
    UINT8     *Dst1;
    EFI_STATUS  Status;
    SCRATCH_DATA *Sd;
    UINT8     *Src;
    UINT8     *Dst;

    Status = EFI_SUCCESS;
    Src = Source;
    Dst = Destination;
    Dst1 = Dst;

    if (ScratchSize < sizeof (SCRATCH_DATA)) {
        return EFI_INVALID_PARAMETER;
    }

    Sd = (SCRATCH_DATA *)Scratch;

    if (SrcSize < 8) {
        return EFI_INVALID_PARAMETER;
    }


    if (SrcSize < CompSize + 8) {
        return EFI_INVALID_PARAMETER;
    }

    Src = Src + 8;

    for (Index = 0; Index < sizeof(SCRATCH_DATA); Index++) {
        ((UINT8*)Sd)[Index] = 0;
    }

    Sd->mBytesRemain = (UINT16)(-1);
    Sd->mSrcBase = Src;
    Sd->mDstBase = Dst;
Sd->mCompSize = CompSize;
Sd->mOrigSize = OrigSize;

// Fill the first two bytes
//FillBuf(Sd, BITBUFSIZ);

while (Sd->mOrigSize > 0) {
    Count = (UINT16) (WNDSIZ < Sd->mOrigSize? WNDSIZ: Sd->mOrigSize);
    Decode (Sd, Count);

    if (Sd->mBadTableFlag != 0) {
        // Something wrong with the source
        // return EFI_INVALID_PARAMETER;
    }

    for (Index = 0; Index < Count; Index ++)
        if (Dst1 < Dst + DstSize)
            *Dst1++ = Sd->mBuffer[Index];
        else
            return EFI_INVALID_PARAMETER;

    Sd->mOrigSize -= Count;
}

if (Sd->mBadTableFlag != 0) {
    Status = EFI_INVALID_PARAMETER;
} else {
    Status = EFI_SUCCESS;
}

return Status;


STATIC
VOID
FillBuf ( /*++

Routine Description:

    Shift mBitBuf NumOfBits left. Read in NumOfBits of bits from source.

Arguments:
Sd - The global scratch data
NumOfBit - The number of bits to shift and read.

Returns: (VOID)
--*/
{
Sd->mBitBuf = (UINT16)(Sd->mBitBuf << NumOfBits);
while (NumOfBits > Sd->mBitCount) {
    Sd->mBitBuf |= (UINT16)(Sd->mSubBitBuf << (NumOfBits = (UINT16)(NumOfBits - Sd->mBitCount)));
    if (Sd->mCompSize > 0) {
        // Get 1 byte into SubBitBuf
        //
        Sd->mCompSize --;
        Sd->mSubBitBuf = 0;
        Sd->mSubBitBuf = Sd->mSrcBase[Sd->mInBuf ++];
        Sd->mBitCount = 8;
    } else {
        Sd->mSubBitBuf = 0;
        Sd->mBitCount = 8;
    }
}
Sd->mBitCount = (UINT16)(Sd->mBitCount - NumOfBits);
Sd->mBitBuf |= Sd->mSubBitBuf >> Sd->mBitCount;
}

STATIC UINT16 GetBits(
    IN SCRATCH_DATA *Sd,
    IN UINT16  NumOfBits
 )
/**<**/

Routine Description:

Get NumOfBits of bits out from mBitBuf. Fill mBitBuf with subsequent NumOfBits of bits from source. Returns NumOfBits of bits that are popped out.

Arguments:

Sd - The global scratch data.
NumOfBits - The number of bits to pop and read.
Returns:

The bits that are popped out.

--*/
{
  UINT16 OutBits;
  OutBits = (UINT16)(Sd->mBitBuf >> (BITBUFSIZ - NumOfBits));
  FillBuf (Sd, NumOfBits);
  return OutBits;
}

STATIC
UINT16
MakeTable (  
  IN SCRATCH_DATA *Sd,  
  IN UINT16   NumOfChar,  
  IN UINT8    *BitLen,  
  IN UINT16   TableBits,  
  OUT UINT16    *Table  
  )
/*++
Routine Description:

Creates Huffman Code mapping table according to code length array.

Arguments:

  Sd    - The global scratch data
  NumOfChar - Number of symbols in the symbol set
  BitLen  - Code length array
  TableBits - The width of the mapping table
  Table   - The table

Returns:

  0     - OK.
  BAD_TABLE - The table is corrupted.

--*/
{
  UINT16 Count[17];
  UINT16 Weight[17];
  UINT16 Start[18];
  UINT16 *p;
  UINT16 k;
  UINT16 i;
  UINT16 Len;
  UINT16 Char;
UINT16 JuBits;
UINT16 Avail;
UINT16 NextCode;
UINT16 Mask;

for (i = 1; i <= 16; i ++) {
    Count[i] = 0;
}

for (i = 0; i < NumOfChar; i++) {
    Count[BitLen[i]]++;
}

Start[1] = 0;
for (i = 1; i <= 16; i ++) {
    Start[i + 1] = (UINT16)(Start[i] + (Count[i] << (16 - i)));
}

if (Start[17] != 0) {/*(1U << 16)*/
    return (UINT16)BAD_TABLE;
}

JuBits = (UINT16)(16 - TableBits);

for (i = 1; i <= TableBits; i ++) {
    Start[i] >>= JuBits;
    Weight[i] = (UINT16)(1U << (TableBits - i));
}

while (i <= 16) {
    Weight[i++] = (UINT16)(1U << (16 - i));
}

i = (UINT16)(Start[TableBits + 1] >> JuBits);

if (i != 0) {
    k = (UINT16)(1U << TableBits);
    while (i != k) {
        Table[i++] = 0;
    }
}

Avail = NumOfChar;
Mask = (UINT16)(1U << (15 - TableBits));

for (Char = 0; Char < NumOfChar; Char++) {
    Len = BitLen[Char];
    if (Len == 0) {
        continue;
    }

if (Len <= TableBits) {
    for (i = Start[Len]; i < NextCode; i ++) {
        Table[i] = Char;
    }
} else {
    k = Start[Len];
    p = &Table[k >> JuBits];
    i = (UINT16)(Len - TableBits);

    while (i != 0) {
        if (*p == 0) {
            Sd->mRight[Avail] = Sd->mLeft[Avail] = 0;
            *p = Avail ++;
        }
        if (k & Mask) {
            p = &Sd->mRight[*p];
        } else {
            p = &Sd->mLeft[*p];
        }

        k <<= 1;
        i --;
    }

    *p = Char;
}

Start[Len] = NextCode;
}

//
// Succeeds
//
return 0;
}

STATIC
UINT16
DecodeP (  
    IN SCRATCH_DATA *Sd
    )
/*++
Routine description:
  Decodes a position value.
Arguments:
Sd - the global scratch data

Returns:

The position value decoded.

--*/
{
    UINT16 Val;
    UINT16 Mask;
    Val = Sd->mPTTable[Sd->mBitBuf >> (BITBUFSIZ - 8)];
    if (Val >= NP) {
        Mask = 1U << (BITBUFSIZ - 1 - 8);
        do {
            if (Sd->mBitBuf & Mask) {
                Val = Sd->mRight[Val];
            } else {
                Val = Sd->mLeft[Val];
            }
            Mask >>= 1;
        } while (Val >= NP);
    }
    //
    // Advance what we have read
    //
    FillBuf (Sd, Sd->mPTLen[Val]);
    if (Val) {
        Val = (UINT16)((1U << (Val - 1)) + GetBits (Sd, (UINT16)(Val - 1)));
    }
    return Val;
}

STATIC
UINT16
ReadPTLen (  
    IN SCRATCH_DATA *Sd,  
    IN UINT16 nn,  
    IN UINT16 nbit,  
    IN UINT16 Special  
)  
/*++

Routine Description

Reads code lengths for the Extra Set or the Position Set
Arguments:

- Sd    - The global scratch data
- nn    - Number of symbols
- nbit  - Number of bits needed to represent nn
- Special - The special symbol that needs to be taken care of

Returns:

- 0     - OK.
- BAD_TABLE - Table is corrupted.

```c
UINT16  n;
UINT16  c;
UINT16  i;
UINT16  Mask;

n = GetBits (Sd, nbit);
if (n == 0) {
    c = GetBits (Sd, nbit);
    for ( i = 0; i < 256; i++) {
        Sd->mPTTable[i] = c;
    }
    for ( i = 0; i < nn; i++) {
        Sd->mPTLen[i] = 0;
    }
    return 0;
}

i = 0;
while (i < n) {
    c = (UINT16)(Sd->mBitBuf >> (BITBUFSIZ - 3));
    if (c == 7) {
        Mask = 1U << (BITBUFSIZ - 1 - 3);
        while (Mask & Sd->mBitBuf) {
            Mask >>= 1;
            c += 1;
        }
    }
    FillBuf (Sd, (UINT16)((c < 7) ? 3 : c - 3));
    Sd->mPTLen [i++] = (UINT8)c;
    if (i == Special) {
```
c = GetBits (Sd, 2);
while ((INT16)(--c) >= 0) {
    Sd->mPTLen[i++] = 0;
}
}
}
while (i < nn) {
    Sd->mPTLen [i++] = 0;
}
return ( MakeTable (Sd, nn, Sd->mPTLen, 8, Sd->mPTTable) );
}

STATIC
VOID
ReadCLen (SCRATCH_DATA *Sd)
/*@++
Routine Description:

Reads code lengths for Char&Len Set.
Arguments:

Sd  - the global scratch data
Returns: (VOID)
--*/
{
    UINT16  n;
    UINT16  c;
    UINT16  i;
    UINT16  Mask;

    n = GetBits(Sd, CBIT);
    if (n == 0) {
        c = GetBits(Sd, CBIT);
        for (i = 0; i < NC; i ++) {
            Sd->mCLen[i] = 0;
        }
        for (i = 0; i < 4096; i ++) {
            Sd->mCTable[i] = c;
        }
        return;
    }
i = 0;
while (i < n) {
    c = Sd->mPTTable[Sd->mBitBuf >> (BITBUFSIZ - 8)];
    if (c >= NT) {
        Mask = 1U << (BITBUFSIZ - 1 - 8);
        do {
            if (Mask & Sd->mBitBuf) {
                c = Sd->mRight[c];
            } else {
                c = Sd->mLeft[c];
            }
            Mask >>= 1;
        }while (c >= NT);
    }
    // Advance what we have read
    // FillBuf (Sd, Sd->mPTLen[c]);
    if (c <= 2) {
        if (c == 0) {
            c = 1;
        } else if (c == 1) {
            c = (UINT16)(GetBits (Sd, 4) + 3);
        } else if (c == 2) {
            c = (UINT16)(GetBits (Sd, CBIT) + 20);
        }
        while ((INT16)(--c) >= 0) {
            Sd->mCLen[i++] = 0;
        }
    } else {
        Sd->mCLen[i++] = (UINT8)(c - 2);
    }
}
while (i < NC) {
    Sd->mCLen[i++] = 0;
}
MakeTable (Sd, NC, Sd->mCLen, 12, Sd->mCTable);
return;
STATIC
UINT16
DecodeC (SCRATCH_DATA *Sd)
/
/**
Routine Description:
    Decode a character/length value.
Arguments:
    Sd  - The global scratch data.
Returns:
    The value decoded.
*/
{
    UINT16 j;
    UINT16 Mask;
    if (Sd->mBlockSize == 0) {
        // Starting a new block
        Sd->mBlockSize = GetBits(Sd, 16);
        Sd->mBadTableFlag = ReadPTLen (Sd, NT, TBIT, 3);
        if (Sd->mBadTableFlag != 0) {
            return 0;
        }
    }
    ReadCLen (Sd);
    Sd->mBadTableFlag = ReadPTLen (Sd, NP, PBIT, (UINT16)(-1));
    if (Sd->mBadTableFlag != 0) {
        return 0;
    }
    Sd->mBlockSize --;
    j = Sd->mCTable[Sd->mBitBuf >> (BITBUFSIZ - 12)];
    if (j >= NC) {
        Mask = 1U << (BITBUFSIZ - 1 - 12);
        do {
            if (Sd->mBitBuf & Mask) {
                j = Sd->mRight[j];
            } else {
                break;
            }
        } while (j >= NC);
    }
}
\[ j = Sd->mLeft[j]; \]

\[
\text{Mask} \gg= 1; \]
\[
} \text{ while } (j \geq NC); \]
\[
} \\
// Advance what we have read 
// FillBuf(Sd, Sd->mCLen[j]); 
\]
\[
return j; \]
\}

\text{STATIC} \\
\text{VOID} \\
\text{Decode (} \\
\text{SCRATCH\_DATA *Sd,} \\
\text{UINT16 NumOfBytes} \\
\text{)} \\
\text{/***/} \\
\text{Routine Description:} \\
\text{Decode NumOfBytes and put the resulting data at starting point of mBuffer.} \\
\text{The buffer is circular.} \\
\text{Arguments:} \\
\text{Sd - The global scratch data} \\
\text{NumOfBytes - Number of bytes to decode} \\
\text{Returns: (VOID)} \\
\text{--*/} \\
\{ \\
\text{UINT16 di;} \\
\text{UINT16 r;} \\
\text{UINT16 c;} \\
\text{r = 0;} \\
\text{di = 0;} \\
\text{Sd->mBytesRemain --;} \\
\text{while ((INT16)(Sd->mBytesRemain) \geq 0) {} \\
\text{Sd->mBuffer[di++] = Sd->mBuffer[Sd->mDataIdx++];} \\
\text{if (Sd->mDataIdx \geq WNDSIZ) { } \\
\text{Sd->mDataIdx -= WNDSIZ; } \\
\text{}} \\
\text{r ++;} \\
\text{if (r \geq NumOfBytes) { } \
\} 
\}
return;
}
Sd->mBytesRemain --;
}

for (;;) {
c = DecodeC (Sd);
if (Sd->mBadTableFlag != 0) {
    return;
}

if (c < 256) {
    //
    // Process an Original character
    //
    Sd->mBuffer[di++] = (UINT8)c;
r ++;
    if (di >= WNDSIZ) {
        return;
    }
} else {
    //
    // Process a Pointer
    //
    c = (UINT16)(c - (UINT8_MAX + 1 - THRESHOLD));
    Sd->mBytesRemain = c;
    Sd->mDataIdx = (r - DecodeP(Sd) - 1) & (WNDSIZ - 1); //Make circular
di = r;
    Sd->mBytesRemain --;
    while (((INT16)(Sd->mBytesRemain) >= 0) {
        Sd->mBuffer[di++] = Sd->mBuffer[Sd->mDataIdx++];
        if (Sd->mDataIdx >= WNDSIZ) {
            Sd->mDataIdx -= WNDSIZ;
        }
        r ++;
        if (di >= WNDSIZ) {
            return;
        }
    }Sd->mBytesRemain --;
}

return;
Appendix J - EFI Byte Code Virtual Machine Opcode List

The following table lists the opcodes for EBC instructions. Note that opcodes only require 6 bits of the opcode byte of EBC instructions. The other two bits are used for other encodings that are dependent on the particular instruction.

Table J-1 EBC Virtual Machine Opcode Summary

<table>
<thead>
<tr>
<th>Opcode</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x00</td>
<td>BREAK [break code]</td>
</tr>
<tr>
<td>0x01</td>
<td>JMP32(cs</td>
</tr>
<tr>
<td></td>
<td>JMP64(cs</td>
</tr>
<tr>
<td>0x02</td>
<td>JMP8(cs</td>
</tr>
<tr>
<td>0x03</td>
<td>CALL32<a href="a">EX</a> [@]R1 {Immed32</td>
</tr>
<tr>
<td></td>
<td>CALL64<a href="a">EX</a> Immed64</td>
</tr>
<tr>
<td>0x04</td>
<td>RET</td>
</tr>
<tr>
<td>0x05</td>
<td>CMP32</td>
</tr>
<tr>
<td>0x06</td>
<td>CMP32</td>
</tr>
<tr>
<td>0x07</td>
<td>CMP32</td>
</tr>
<tr>
<td>0x08</td>
<td>CMP32</td>
</tr>
<tr>
<td>0x09</td>
<td>CMP32</td>
</tr>
<tr>
<td>0x0A</td>
<td>NOT32</td>
</tr>
<tr>
<td>0x0B</td>
<td>NEG32</td>
</tr>
<tr>
<td>0x0C</td>
<td>ADD32</td>
</tr>
<tr>
<td>0x0D</td>
<td>SUB32</td>
</tr>
<tr>
<td>0x0E</td>
<td>MUL32</td>
</tr>
<tr>
<td>0x0F</td>
<td>MULU32</td>
</tr>
<tr>
<td>0x10</td>
<td>DIV32</td>
</tr>
<tr>
<td>0x11</td>
<td>DIVU32</td>
</tr>
<tr>
<td>0x12</td>
<td>MOD32</td>
</tr>
<tr>
<td>0x13</td>
<td>MODU32</td>
</tr>
<tr>
<td>0x14</td>
<td>AND32</td>
</tr>
<tr>
<td>0x15</td>
<td>OR32</td>
</tr>
<tr>
<td>0x16</td>
<td>XOR32</td>
</tr>
<tr>
<td>0x17</td>
<td>SHL32</td>
</tr>
<tr>
<td>0x18</td>
<td>SHR32</td>
</tr>
<tr>
<td>Opcode</td>
<td>Description</td>
</tr>
<tr>
<td>--------</td>
<td>-------------</td>
</tr>
<tr>
<td>0x19</td>
<td>ASHR[32</td>
</tr>
<tr>
<td>0x1A</td>
<td>EXTND[32</td>
</tr>
<tr>
<td>0x1B</td>
<td>EXTND[32</td>
</tr>
<tr>
<td>0x1C</td>
<td>EXTND[32</td>
</tr>
<tr>
<td>0x1D</td>
<td>MOVbw [@]R1 {Index16}, [@]R2 {Index16}</td>
</tr>
<tr>
<td>0x1E</td>
<td>MOVww [@]R1 {Index16}, [@]R2 {Index16}</td>
</tr>
<tr>
<td>0x1F</td>
<td>MOVdw [@]R1 {Index16}, [@]R2 {Index16}</td>
</tr>
<tr>
<td>0x20</td>
<td>MOVqw [@]R1 {Index16}, [@]R2 {Index16}</td>
</tr>
<tr>
<td>0x21</td>
<td>MOVbd [@]R1 {Index32}, [@]R2 {Index32}</td>
</tr>
<tr>
<td>0x22</td>
<td>MOVwd [@]R1 {Index32}, [@]R2 {Index32}</td>
</tr>
<tr>
<td>0x23</td>
<td>MOVdd [@]R1 {Index32}, [@]R2 {Index32}</td>
</tr>
<tr>
<td>0x24</td>
<td>MOVqd [@]R1 {Index32}, [@]R2 {Index32}</td>
</tr>
<tr>
<td>0x25</td>
<td>MOVsnw [@]R1 {Index16}, [@]R2 {Index16</td>
</tr>
<tr>
<td>0x26</td>
<td>MOVsnd [@]R1 {Index32}, [@]R2 {Index32</td>
</tr>
<tr>
<td>0x27</td>
<td>Reserved</td>
</tr>
<tr>
<td>0x28</td>
<td>MOVqq [@]R1 {Index64}, [@]R2 {Index64}</td>
</tr>
<tr>
<td>0x29</td>
<td>LOADSP [Flags], R2</td>
</tr>
<tr>
<td>0x2A</td>
<td>STORESP R1, [IP</td>
</tr>
<tr>
<td>0x2B</td>
<td>PUSH[32</td>
</tr>
<tr>
<td>0x2C</td>
<td>POP[32</td>
</tr>
<tr>
<td>0x2D</td>
<td>CMPI[32</td>
</tr>
<tr>
<td>0x2E</td>
<td>CMPI[32</td>
</tr>
<tr>
<td>0x2F</td>
<td>CMPI[32</td>
</tr>
<tr>
<td>0x30</td>
<td>CMPI[32</td>
</tr>
<tr>
<td>0x31</td>
<td>CMPI[32</td>
</tr>
<tr>
<td>0x32</td>
<td>MOVnw [@]R1 {Index16}, [@]R2 {Index16}</td>
</tr>
<tr>
<td>0x33</td>
<td>MOVnd [@]R1 {Index32}, [@]R2 {Index32}</td>
</tr>
<tr>
<td>0x34</td>
<td>Reserved</td>
</tr>
<tr>
<td>0x35</td>
<td>PUSHn1 [@]R1 {Index16</td>
</tr>
<tr>
<td>0x36</td>
<td>POPn [@]R1 {Index16</td>
</tr>
<tr>
<td>0x37</td>
<td>MOV[0</td>
</tr>
<tr>
<td>0x38</td>
<td>MOVIn[0</td>
</tr>
<tr>
<td>0x39</td>
<td>MOVREL[w</td>
</tr>
<tr>
<td>Opcode</td>
<td>Description</td>
</tr>
<tr>
<td>--------</td>
<td>-------------</td>
</tr>
<tr>
<td>0x3A</td>
<td>Reserved</td>
</tr>
<tr>
<td>0x3B</td>
<td>Reserved</td>
</tr>
<tr>
<td>0x3C</td>
<td>Reserved</td>
</tr>
<tr>
<td>0x3D</td>
<td>Reserved</td>
</tr>
<tr>
<td>0x3E</td>
<td>Reserved</td>
</tr>
<tr>
<td>0x3F</td>
<td>Reserved</td>
</tr>
</tbody>
</table>
Appendix K - Alphabetic Function Lists

This appendix was redacted in version 2.6.
Appendix L - EFI 1.10 Protocol Changes and Deprecation List

L.1 Protocol and GUID Name Changes from EFI 1.10

This appendix lists the Protocol, GUID, and revision identifier name changes and the deprecated protocols compared to the EFI Specification 1.10. The protocols listed are not Runtime, Reentrant or MP Safe. Protocols are listed by EFI 1.10 name.

For protocols in the table whose TPL is not <= TPL_NOTIFY:

This function must be called at a TPL level less then or equal to %%%%. %%% is TPL_CALLBACK or TPL_APPLICATION. The <= is done via text.

Table L-1 Protocol Name changes

<table>
<thead>
<tr>
<th>EFI 1.10 Protocol Name</th>
<th>UEFI Specification Protocol Name</th>
</tr>
</thead>
<tbody>
<tr>
<td>EFI_LOADED_IMAGE</td>
<td>EFI_LOADED_IMAGE_PROTOCOL</td>
</tr>
<tr>
<td>TPL</td>
<td>&lt;= TPL_NOTIFY</td>
</tr>
<tr>
<td>New GUID name</td>
<td>EFI_LOADED_IMAGE_PROTOCOL_GUID</td>
</tr>
<tr>
<td>EFI_DEVICE_PATH</td>
<td>EFI_DEVICE_PATH_PROTOCOL</td>
</tr>
<tr>
<td>TPL</td>
<td>&lt;= TPL_NOTIFY</td>
</tr>
<tr>
<td>New GUID name</td>
<td>EFI_DEVICE_PATH_PROTOCOL_GUID</td>
</tr>
<tr>
<td>SIMPLE_INPUT_INTERFACE</td>
<td>EFI_SIMPLE_INPUT_PROTOCOL</td>
</tr>
<tr>
<td>TPL</td>
<td>&lt;= TPL_APPLICATION</td>
</tr>
<tr>
<td>New GUID name</td>
<td>EFI_SIMPLE_INPUT_PROTOCOL_GUID</td>
</tr>
<tr>
<td>SIMPLE_TEXT_OUTPUT_INTERFACE</td>
<td>EFI_SIMPLE_TEXT_OUTPUT_PROTOCOL</td>
</tr>
<tr>
<td>TPL</td>
<td>&lt;= TPL_CALLBACK</td>
</tr>
<tr>
<td>New GUID name</td>
<td>EFI_SIMPLE_TEXT_OUTPUT_PROTOCOL_GUID</td>
</tr>
<tr>
<td>SERIAL_IO_INTERFACE</td>
<td>EFI_SERIAL_IO_PROTOCOL</td>
</tr>
<tr>
<td>TPL</td>
<td>&lt;= TPL_CALLBACK</td>
</tr>
<tr>
<td>New GUID name</td>
<td>EFI_SERIAL_IO_PROTOCOL_GUID</td>
</tr>
<tr>
<td>EFI_LOAD_FILE_INTERFACE</td>
<td>EFI_LOAD_FILE_PROTOCOL</td>
</tr>
<tr>
<td>TPL</td>
<td>&lt;= TPL_NOTIFY</td>
</tr>
<tr>
<td>New GUID name</td>
<td>EFI_LOAD_FILE_PROTOCOL_GUID</td>
</tr>
<tr>
<td>EFI_FILE_IO_INTERFACE</td>
<td>EFI_SIMPLE_FILE_SYSTEM_PROTOCOL</td>
</tr>
<tr>
<td>TPL</td>
<td>&lt;= TPL_CALLBACK</td>
</tr>
<tr>
<td>New GUID name</td>
<td>EFI_LOAD_FILE_PROTOCOL_GUID</td>
</tr>
<tr>
<td>EFI_FILE</td>
<td>EFI_FILE_PROTOCOL</td>
</tr>
<tr>
<td>TPL</td>
<td>&lt;= TPL_CALLBACK</td>
</tr>
<tr>
<td>New GUID name</td>
<td>EFI_FILE_PROTOCOL</td>
</tr>
<tr>
<td>EFI_DISK_IO</td>
<td>EFI_DISK_IO_PROTOCOL</td>
</tr>
<tr>
<td>TPL</td>
<td>&lt;= TPL_CALLBACK</td>
</tr>
<tr>
<td>New GUID name</td>
<td>EFI_DISK_IO_PROTOCOL</td>
</tr>
</tbody>
</table>
## Table L-2 Revision Identifier Name Changes

<table>
<thead>
<tr>
<th>EFI 1.10 Revision Identifier Name</th>
<th>UEFI Specification Revision Identifier Name</th>
</tr>
</thead>
<tbody>
<tr>
<td>EFI_LOADED_IMAGE_INFORMATION_REVISION</td>
<td>EFI_LOADED_IMAGE_PROTOCOL_REVISION</td>
</tr>
<tr>
<td>SERIAL_IO_INTERFACE_REVISION</td>
<td>EFI_SERIAL_IO_PROTOCOL_REVISION</td>
</tr>
<tr>
<td>EFI_FILE_IO_INTERFACE_REVISION</td>
<td>EFI_SIMPLE_FILE_SYSTEM_PROTOCOL_REVISION</td>
</tr>
<tr>
<td>EFI_FILE_REVISION</td>
<td>EFI_FILE_PROTOCOL_REVISION</td>
</tr>
<tr>
<td>EFI_DISK_IO_INTERFACE_REVISION</td>
<td>EFI_DISK_IO_PROTOCOL_REVISION</td>
</tr>
<tr>
<td>EFI_BLOCK_IO_INTERFACE_REVISION</td>
<td>EFI_BLOCK_IO_PROTOCOL_REVISION</td>
</tr>
<tr>
<td>EFI_SIMPLE_NETWORK_INTERFACE_REVISION</td>
<td>EFI_SIMPLE_NETWORK_PROTOCOL_REVISION</td>
</tr>
<tr>
<td>EFI_NETWORK_INTERFACE_REVISION</td>
<td>EFI_NETWORK_INTERFACE_PROTOCOL_REVISION</td>
</tr>
<tr>
<td>EFI_PXE_BASE_CODE_INTERFACE_REVISION</td>
<td>EFI_PXE_BASE_CODE_PROTOCOL_REVISION</td>
</tr>
</tbody>
</table>

### Table L-2 Revision Identifier Name Changes

<table>
<thead>
<tr>
<th>EFI 1.10 Protocol Name</th>
<th>UEFI Specification Protocol Name</th>
</tr>
</thead>
<tbody>
<tr>
<td>EFI_BLOCK_IO</td>
<td>EFI_BLOCK_IO_PROTOCOL</td>
</tr>
<tr>
<td>TPL &lt;= TPL_CALLBACK</td>
<td></td>
</tr>
<tr>
<td>New GUID name</td>
<td>EFI_BLOCK_IO_PROTOCOL_GUID</td>
</tr>
<tr>
<td>UNICODE_COLLATION_INTERFACE</td>
<td>EFI_UNICODE_COLLATION_PROTOCOL</td>
</tr>
<tr>
<td>TPL &lt;= TPL_NOTIFY</td>
<td></td>
</tr>
<tr>
<td>New GUID name</td>
<td>EFI_UNICODE_COLLATION_PROTOCOL_GUID</td>
</tr>
<tr>
<td>EFI_SIMPLE_NETWORK</td>
<td>EFI_SIMPLE_NETWORK_PROTOCOL</td>
</tr>
<tr>
<td>TPL &lt;= TPL_CALLBACK</td>
<td></td>
</tr>
<tr>
<td>New GUID name</td>
<td>EFI_SIMPLE_NETWORK_PROTOCOL_GUID</td>
</tr>
<tr>
<td>EFI_NETWORK_INTERFACE_IDENTIFIER INTERFACE</td>
<td>EFI_NETWORK_INTERFACE_IDENTIFIER_PROTOCOL</td>
</tr>
<tr>
<td>TPL &lt;= TPL_NOTIFY</td>
<td></td>
</tr>
<tr>
<td>New GUID name</td>
<td>EFI_NETWORK_INTERFACE_IDENTIFIER_PROTOCOL_GUID</td>
</tr>
<tr>
<td>EFI_PXE_BASE_CODE</td>
<td>EFI_PXE_BASE_CODE_PROTOCOL</td>
</tr>
<tr>
<td>TPL &lt;= TPL_NOTIFY</td>
<td></td>
</tr>
<tr>
<td>New GUID name</td>
<td>EFI_PXE_BASE_CODE_PROTOCOL_GUID</td>
</tr>
<tr>
<td>EFI_PXE_BASE_CODE_CALLBACK INTERFACE</td>
<td>EFI_PXE_BASE_CODE_CALLBACK_PROTOCOL</td>
</tr>
<tr>
<td>TPL &lt;= TPL_NOTIFY</td>
<td></td>
</tr>
<tr>
<td>New GUID name</td>
<td>EFI_PXE_BASE_CODE_CALLBACK_PROTOCOL_GUID</td>
</tr>
<tr>
<td>EFI_DEVICE_IO_INTERFACE</td>
<td>EFI_DEVICE_IO_PROTOCOL</td>
</tr>
<tr>
<td>TPL &lt;= TPL_NOTIFY</td>
<td></td>
</tr>
<tr>
<td>New GUID name</td>
<td>EFI_DEVICE_IO_PROTOCOL_GUID</td>
</tr>
</tbody>
</table>

### EFI 1.10 Protocol Name UEFI Specification Protocol Name

- EFI_BLOCK_IO
- EFI_BLOCK_IO_PROTOCOL
- EFI_UNICODE_COLLATION_PROTOCOL
- EFI_UNICODE_COLLATION_PROTOCOL_GUID
- EFI_SIMPLE_NETWORK
- EFI_SIMPLE_NETWORK_PROTOCOL
- EFI_SIMPLE_NETWORK_PROTOCOL_GUID
- EFI_NETWORK_INTERFACE_IDENTIFIER_INTERFACE
- EFI_NETWORK_INTERFACE_IDENTIFIER_PROTOCOL
- EFI_NETWORK_INTERFACE_IDENTIFIER_PROTOCOL_GUID
- EFI_PXE_BASE_CODE
- EFI_PXE_BASE_CODE_PROTOCOL
- EFI_PXE_BASE_CODE_CALLBACK
- EFI_PXE_BASE_CODE_CALLBACK_PROTOCOL
- EFI_DEVICE_IO_INTERFACE
- EFI_DEVICE_IO_PROTOCOL
- EFI_DEVICE_IO_PROTOCOL_GUID

### EFI 1.10 Revision Identifier Name UEFI Specification Revision Identifier Name

- EFI_LOADED_IMAGE_INFORMATION_REVISION
- EFI_LOADED_IMAGE_PROTOCOL_REVISION
- EFI_SERIAL_IO_PROTOCOL_REVISION
- EFI_SIMPLE_FILE_SYSTEM_PROTOCOL_REVISION
- EFI_FILE_PROTOCOL_REVISION
- EFI_DISK_IO_PROTOCOL_REVISION
- EFI_BLOCK_IO_PROTOCOL_REVISION
- EFI_SIMPLE_NETWORK_PROTOCOL_REVISION
- EFI_NETWORK_INTERFACE_IDENTIFIER_PROTOCOL_REVISION
- EFI_PXE_BASE_CODE_PROTOCOL_REVISION
- EFI_PXE_BASE_CODE_CALLBACK_PROTOCOL_REVISION

---

UEFI Forum, Inc.  March 2021
L.2 Deprecated Protocols

**Device I/O Protocol** – The support of the Device I/O Protocol (see EFI 1.1 Chapter 18) has been replaced by the use of the **PCI Root Bridge I/O** protocols which are described in **Section 14.2** of the UEFI Specification. Note: certain “legacy” EFI applications such as some of the ones that reside in the EFI Toolkit assume the presence of Device I/O.

**UGA I/O + UGA Draw Protocol** – The support of the UGA * Protocols (see EFI 1.1 Section 10.7) have been replaced by the use of the **EFI Graphics Output Protocol** described in **Section 12** of the UEFI Specification.

**USB Host Controller Protocol** (version that existed for EFI 1.1) – The support of the USB Host Controller Protocol (see EFI 1.1 Section 14.1) has been replaced by the use of a UEFI instance that covers both USB 1.1 and USB 2.0 support, and is described in **Section 17** of the UEFI Specification. It replaces the pre-existing protocol definition.

**SCSI Passthru Protocol** – The support of the SCSI Passthru Protocol (see EFI 1.1 Section 13.1) has been replaced by the use of the **Extended SCSI Passthru Protocol** which is described in Chapter **Section 15.7** of the UEFI Specification.

**BIS Protocol** – Remains as an optional protocol.

**Driver Configuration Protocol** - the **EFI_DRIVER_CONFIGURATION_PROTOCOL** has been removed.
Appendix M - Formats — Language Codes and Language Code Arrays

This appendix lists the formats for language codes and language code arrays.

M.1 Specifying individual language codes

The preferred representation of a language code is done via an RFC 4646 language code identifier*.

Table M-1 Alias codes supported in addition to RFC 4646

<table>
<thead>
<tr>
<th>RFC string</th>
<th>Supported Alias String</th>
</tr>
</thead>
<tbody>
<tr>
<td>zh-Hans</td>
<td>zh-chs</td>
</tr>
<tr>
<td>zh-Hant</td>
<td>zh-cht</td>
</tr>
</tbody>
</table>

An RFC 4646 language code is represented as a null-terminated ASCII string.

An RFC 4646 language string must be constructed according to the tag creation rules in section 2.3 of RFC 4646. For example, when constructing the primary language tag for a locale identifier, if a 2 character ISO 639-1 language code exists along with a 3 character ISO 639-2 language code, then the ISO 639-1 language code must be used. Further, if an ISO 639-1 tag does not exist, then the ISO 639-2/T (Terminology) tag must be for the primary locale before an ISO 639-2/B (Bibliographic) tag may be used. See RFC 4646 for a complete discussion of this topic.

To provide backwards compatibility with preexisting EFI 1.10 drivers, a UEFI platforms may support deprecated protocols which represent languages in the ISO 639-2 format. This includes the following protocols: `UNICODE_COLLATION_INTERFACE`, `EFI_DRIVER_CONFIGURATION_PROTOCOL`, `EFI_DRIVER_DIAGNOSTICS_PROTOCOL`, and `EFI_COMPONENT_NAME_PROTOCOL`. The deprecated `LangCodes` and `Lang` global variables may also be supported by a platform for backwards compatibility.

M.1.1 Specifying language code arrays:

Native RFC 4646 format array:

An array of RFC 4646 character codes is represented as a NULL terminated char8 array of RFC 4646 language code strings. Each of these strings is delimited by a semicolon (";") character. For example, an array of US English and Traditional Chinese would be represented as the NULL-terminated string “en-us;zh-Hant”.
Appendix N - Common Platform Error Record

N.1 Introduction
This appendix describes the common platform error record format for representing platform hardware errors.

N.2 Format
The general format of the common platform error record is illustrated in Figure N-1. The record consists of a header; followed by one or more section descriptors; and for each descriptor, an associated section which may contain either error or informational data.

N.2.1 Record Header
The record header includes information which uniquely identifies a hardware error record on a given system. The contents of the record header are described in Table N-1. The header is immediately followed by an array of one or more section descriptors. Sections may be either error sections, which contain error information retrieved from hardware, or they may be informational sections, which contain contextual information relevant to the error. An error record must contain at least one section.
## Table N-1 Error record header

<table>
<thead>
<tr>
<th>Mnemonic</th>
<th>Byte Offset</th>
<th>Byte Length</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Signature Start</td>
<td>0</td>
<td>4</td>
<td>ASCII 4-character array &quot;CPER&quot; (0x43,0x50,0x45,0x52). Identifies this structure as a hardware error record.</td>
</tr>
<tr>
<td>Revision</td>
<td>4</td>
<td>2</td>
<td>This is a 2-byte field representing a major and minor version number for the error record definition in BCD format. The interpretation of the major and minor version number is as follows:   · Byte 0 – Minor (01): An increase in this revision indicates that changes to the headers and sections are backward compatible with software that use earlier revisions. Addition of new GUID types, errata fixes or clarifications are covered by a bump up. · Byte 1 – Major (01): An increase in this revision indicates that the changes are not backward compatible from a software perspective.</td>
</tr>
<tr>
<td>Signature End</td>
<td>6</td>
<td>4</td>
<td>Must be 0xFFFFFFFF</td>
</tr>
<tr>
<td>Section Count</td>
<td>10</td>
<td>2</td>
<td>This field indicates the number of valid sections associated with the record, corresponding to each of the following section descriptors.</td>
</tr>
<tr>
<td>Error Severity</td>
<td>12</td>
<td>4</td>
<td>Indicates the severity of the error condition. The severity of the error record corresponds to the most severe error section.   · 0 - Recoverable (also called non-fatal uncorrected)   · 1 - Fatal   · 2 - Corrected   · 3 - Informational   · All other values are reserved. Note that severity of &quot;Informational&quot; indicates that the record could be safely ignored by error handling software.</td>
</tr>
<tr>
<td>Validation Bits</td>
<td>16</td>
<td>4</td>
<td>This field indicates the validity of the following fields:   · Bit 0 – If 1, the PlatformID field contains valid information   · Bit 1 – If 1, the TimeStamp field contains valid information   · Bit2 – If 1, the PartitionID field contains valid information   · Bits 3-31: Reserved, must be zero.</td>
</tr>
<tr>
<td>Record Length</td>
<td>20</td>
<td>4</td>
<td>Indicates the size of the actual error record, including the size of the record header, all section descriptors, and section bodies. The size may include extra buffer space to allow for the dynamic addition of error sections descriptors and bodies.</td>
</tr>
</tbody>
</table>
The timestamp correlates to the time when the error information was collected by the system software and may not necessarily represent the time of the error event. The timestamp contains the local time in BCD format.  
- Byte 7 – Byte 0:  
  - Byte 7: Timestamp is precise if this bit is set and correlates to the time of the error event.  
  - Bit 7:1: Reserved  
  - Byte 4: Day  
  - Byte 5: Month  
  - Byte 6: Year  
  - Byte 7: Century

This field uniquely identifies the platform with a GUID. The platform’s SMBIOS UUID should be used to populate this field. Error analysis software may use this value to uniquely identify a platform.

If the platform has multiple software partitions, system software may associate a GUID with the partition on which the error occurred.

This field contains a GUID indicating the creator of the error record. This value may be overwritten by subsequent owners of the record.

<table>
<thead>
<tr>
<th>Mnemonic</th>
<th>Byte Offset</th>
<th>Byte Length</th>
<th>Description</th>
</tr>
</thead>
</table>
| Timestamp     | 24          | 8           | The timestamp correlates to the time when the error information was collected by the system software and may not necessarily represent the time of the error event. The timestamp contains the local time in BCD format.  
- Byte 7 – Byte 0:  
  - Byte 7: Timestamp is precise if this bit is set and correlates to the time of the error event.  
  - Bit 7:1: Reserved  
  - Byte 4: Day  
  - Byte 5: Month  
  - Byte 6: Year  
  - Byte 7: Century |
| Platform ID   | 32          | 16          | This field uniquely identifies the platform with a GUID. The platform’s SMBIOS UUID should be used to populate this field. Error analysis software may use this value to uniquely identify a platform. |
| Partition ID  | 48          | 16          | If the platform has multiple software partitions, system software may associate a GUID with the partition on which the error occurred. |
| Creator ID    | 64          | 16          | This field contains a GUID indicating the creator of the error record. This value may be overwritten by subsequent owners of the record. |
### Notification Type

This field holds a pre-assigned GUID value indicating the record association with an error event notification type. The defined types are:

- **CMC**
  
  `{0x2DCE8BB1, 0xBDD7, 0x450e, {0xB9, 0xAD, 0x9C, 0xF4, 0xEB, 0xD4, 0xF8, 0x90}}`

- **CPE**
  
  `{0x4E292F96, 0xD843, 0x4a55, {0xA8, 0xC2, 0xD4, 0x81, 0xF2, 0x7E, 0xBE, 0xEE}}`

- **MCE**
  
  `{0xE8F56FFE, 0x919C, 0x4cc5, {0xBA, 0x88, 0x65, 0xAB, 0xE1, 0x49, 0x13, 0xBB}}`

- **PCIe**
  
  `{0xCF93C01F, 0x1A16, 0x4dfc, {0xB8, 0xBC, 0x9C, 0x4D, 0xAF, 0x67, 0xC1, 0x04}}`

- **INIT**
  
  `{0xCC5263E8, 0x9308, 0x454a, {0x89, 0xD0, 0x34, 0x0B, 0xD3, 0x9B, 0xC9, 0x8E}}`

- **NMI**
  
  `{0x5BAD89FF, 0xB7E6, 0x42c9, {0xA8, 0xCF, 0x24, 0x85, 0xD6, 0xE9, 0x8A}}`

- **Boot**
  
  `{0x3D61A466, 0xAB40, 0x409a, {0xA6, 0x98, 0xF3, 0x62, 0xD4, 0x64, 0xB3, 0x8F}}`

- **DMAr**
  
  `{0x667DD791, 0x6C6B3, 0x4c27, {0xA8, 0x6B, 0x0F, 0x8E, 0x72, 0x2D, 0xE8, 0x41}}`

- **SEA**
  
  `{0x9A78788A, 0xBBE8, 0x11E4, {0x80, 0x9E, 0x67, 0x61, 0x1E, 0x5D, 0x46, 0xB0}}`

- **SEI**
  
  `{0x5C284CB1, 0xB0AE, 0x4E87, {0xA3, 0x22, 0xB0, 0x4C, 0x85, 0x62, 0x43, 0x23}}`

- **PEI**
  
  `{0x09A9D5AC, 0x5204, 0x41DF, 0x49A3, {0xB4, 0x4D, 0x8F, 0xB0, 0xDB, 0x30, 0x41, 0xF6}}`

### Record ID

This value, when combined with the Creator ID, uniquely identifies the error record across other error records on a given system.

<table>
<thead>
<tr>
<th>Mnemonic</th>
<th>Byte Offset</th>
<th>Byte Length</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Notification Type</td>
<td>80</td>
<td>16</td>
<td>This field holds a pre-assigned GUID value indicating the record association with an error event notification type. The defined types are: CMC <code>{0x2DCE8BB1, 0xBDD7, 0x450e, {0xB9, 0xAD, 0x9C, 0xF4, 0xEB, 0xD4, 0xF8, 0x90}}</code> CPE <code>{0x4E292F96, 0xD843, 0x4a55, {0xA8, 0xC2, 0xD4, 0x81, 0xF2, 0x7E, 0xBE, 0xEE}}</code> MCE <code>{0xE8F56FFE, 0x919C, 0x4cc5, {0xBA, 0x88, 0x65, 0xAB, 0xE1, 0x49, 0x13, 0xBB}}</code> PCIe <code>{0xCF93C01F, 0x1A16, 0x4dfc, {0xB8, 0xBC, 0x9C, 0x4D, 0xAF, 0x67, 0xC1, 0x04}}</code> INIT <code>{0xCC5263E8, 0x9308, 0x454a, {0x89, 0xD0, 0x34, 0x0B, 0xD3, 0x9B, 0xC9, 0x8E}}</code> NMI <code>{0x5BAD89FF, 0xB7E6, 0x42c9, {0xA8, 0xCF, 0x24, 0x85, 0xD6, 0xE9, 0x8A}}</code> Boot <code>{0x3D61A466, 0xAB40, 0x409a, {0xA6, 0x98, 0xF3, 0x62, 0xD4, 0x64, 0xB3, 0x8F}}</code> DMAr <code>{0x667DD791, 0x6C6B3, 0x4c27, {0xA8, 0x6B, 0x0F, 0x8E, 0x72, 0x2D, 0xE8, 0x41}}</code> SEA <code>{0x9A78788A, 0xBBE8, 0x11E4, {0x80, 0x9E, 0x67, 0x61, 0x1E, 0x5D, 0x46, 0xB0}}</code> SEI <code>{0x5C284CB1, 0xB0AE, 0x4E87, {0xA3, 0x22, 0xB0, 0x4C, 0x85, 0x62, 0x43, 0x23}}</code> PEI <code>{0x09A9D5AC, 0x5204, 0x41DF, 0x49A3, {0xB4, 0x4D, 0x8F, 0xB0, 0xDB, 0x30, 0x41, 0xF6}}</code></td>
</tr>
</tbody>
</table>
Table N-2.Error Record Header Flags

<table>
<thead>
<tr>
<th>Value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>HW_ERROR_FLAGS_RECOVERED: Qualifies an error condition as one that has been recovered by system software.</td>
</tr>
<tr>
<td>2</td>
<td>HW_ERROR_FLAGS_PREERR: Qualifies an error condition as one that occurred during a previous session. For instance, if the OS detects an error and determines that the system must be reset; it will save the error record before stopping the system. Upon restarting the OS marks the error record with this flag to know that the error is not live.</td>
</tr>
<tr>
<td>4</td>
<td>HW_ERROR_FLAGS_SIMULATED: Qualifies an error condition as one that was intentionally caused. This allows system software to recognize errors that are injected as a means of validating or testing error handling mechanisms.</td>
</tr>
</tbody>
</table>

**N.2.1.1 Notification Type**

A notification type identifies the mechanism by which an error event is reported to system software. This information helps consumers of error information (e.g. management applications or humans) by identifying the source of the error information. This allows, for instance, all CMC error log entries to be filtered from an error event log.

Listed below are the standard notification types. Each standard notification type is identified by a GUID. For error notification types that do not conform to one of the standard types, a platform-specific GUID may be defined to identify the notification type.

- **Machine Check Exception (MCE):** {0xE8F56FFE, 0x919C, 0x4cc5, {0xBA, 0x88, 0x65, 0xAB, 0xE1, 0x49, 0x13, 0xBB}}
  A Machine Check Exception is a processor-generated exception class interrupt used to system software of the presence of a fatal or recoverable error condition.
- **Corrected Machine Check (CMC):** {0x2DCE8BB1, 0xBDD7, 0x450e, {0xB9, 0xAD, 0x9C, 0xF4,0xEB, 0xD4, 0xF8, 0x90}}
  Corrected Machine Checks identify error conditions that have been corrected by hardware or system firmware. CMCs are reported by the processor and may be reported via interrupt or by polling error status registers.
• Corrected Platform Error (CPE): {0x4E292F96, 0xD843, 0x4a55, {0xA8, 0xC2, 0xD4, 0x81, 0xF2, 0x7E, 0xBE, 0xEE}}
Corrected Platform Errors identify corrected errors from the platform (i.e., external memory controller, system bus, etc.). CPEs can be reported via interrupt or by polling error status registers.

• Non-Maskable Interrupt (NMI): {0x5BAD89FF, 0xB7E6, 0x42c9, {0x81, 0x4A, 0xCF, 0x24, 0x85, 0xD6, 0xE9, 0x8A}}
Non-Maskable Interrupts are used on X64 platforms to report fatal or recoverable platform error conditions. NMIs are reported via interrupt vector 2 on IA32 and X64 processor architecture platforms.

• PCI Express Error (Pcie): {0xCF93C01F, 0x1A16, 0xdfc, {0xB8, 0xBC, 0x9C, 0x4D, 0xAF, 0x67, 0xC1, 0x04}}
See the PCI Express standard v1.1 for details regarding PCI Express Error Reporting. This notification type identifies errors that were reported to the system via an interrupt on a PCI Express root port.

• INIT Record (INIT): {0xCC5263E8, 0x9308, 0x454a, {0x89, 0xD0, 0x34, 0x0B, 0xD3, 0x9B, 0xC9, 0x8E}}
IPF Platforms optionally implement a mechanism (switch or button on the chassis) by which an operator may reset a system and have the system generate an INIT error record. This error record is documented in the IPF SAL specification. System software retrieves an INIT error record by querying the SAL for existing INIT records.

• BOOT Error Record (BOOT): {0x3D61A466, 0xAB40, 0x409a, {0xA6, 0x98, 0xF3, 0x62, 0xD4, 0x64, 0xB3, 0x8F}}
The BOOT Notification Type represents error conditions which are unhandled by system software and which result in a system shutdown/reset. System software retrieves a BOOT error record during boot by querying the platform for existing BOOT records. As an example, consider an x64 platform which implements a service processor. In some scenarios, the service processor may detect that the system is either hung or is in such a state that it cannot safely proceed without risking data corruption. In such a scenario the service processor may record some minimal error information in its system event log (SEL) and unilaterally reset the machine without notifying the OS or other system software. In such scenarios, system software is unaware of the condition that caused the system reset. A BOOT error record would contain information that describes the error condition that led to the reset so system software can log the information and use it for health monitoring.

• DMA Remapping Error (DMAr): {0x667DD791, 0xC6B3, 0x4c27, {0x8A, 0x6B, 0xOF, 0x8E, 0x72, 0x2D, 0xEB, 0x41}}
The DMA Remapping Notification Type identifies fault conditions generated by the DMAr unit when processing un-translated, translation and translated DMA requests. The fault conditions are reported to the system using a message signaled interrupt.

• Synchronous External Abort (SEA): {0x9A78788A, 0xBBE8, 0x11E4, {0x80, 0x9E, 0x67, 0x61, 0x1E, 0x5D, 0x46, 0x80}}
Synchronous External Aborts represent precise processor error conditions on ARM systems (uncorrectable and/or recoverable) as described in D3.5 of the ARMv8 ARM reference manual. This notification may be triggered by one of the following scenarios: cache parity error, cache ECC error, external bus error, micro-architectural error, data poisoning, and other platform errors.
• SError Interrupt (SEI): {0x5C284C81, 0xB0AE, 0x4E87, {0xA3, 0x22, 0xB0, 0x4C, 0x85, 0x62, 0x43, 0x23}}
  SError Interrupts represent asynchronous **imprecise** (or possibly precise) processor error conditions on ARM systems (corrected, uncorrectable, and recoverable) as described in D3.5 of the ARM ARM reference manual. This notification may be triggered by one of the following scenarios: cache parity error, cache ECC error, external bus error, micro-architectural error, data poisoning, and other platform errors.

• Platform Error Interrupt (PEI): {0x09A9D5AC, 0x5204, 0x4214, {0x96, 0xE5, 0x94, 0x99, 0x2E, 0x75, 0x2B, 0xCD}}
  Platform Error Interrupt represent asynchronous **imprecise** platform error conditions on ARM systems that may be triggered by the following scenarios: system memory ECC error, ECC errors in system cache (e.g. shared high-level caches), vendor specific chip errors, external platform errors.

• Compute Express Link (CXL) Component: {0x69293BC9, 0x41DF, 0x49A3 {0xB4, 0xBD, 0x4F, 0xB0, 0xDB, 0x30, 0x41, 0xF6}}
  This Notification Type identifies errors that were reported to the system by CXL components that support error reporting via the CXL RAS Mailbox interface. See the CXL Specification, Rev 2.0 or later, for details regarding CXL Error Reporting.

### N.2.1.2 Error Status

The error status definition provides the capability to abstract information from implementation-specific error registers into generic error codes.

#### Table N-3 Error Status Fields

<table>
<thead>
<tr>
<th>Bit Position</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>7:0</td>
<td>Reserved</td>
</tr>
<tr>
<td>15:8</td>
<td>Encoded value for the Error_Type. See Table 20 Error Types for details.</td>
</tr>
<tr>
<td>16</td>
<td>Address: Error was detected on the address signals or on the address portion of the transaction.</td>
</tr>
<tr>
<td>17</td>
<td>Control: Error was detected on the control signals or in the control portion of the transaction.</td>
</tr>
<tr>
<td>18</td>
<td>Data: Error was detected on the data signals or in the data portion of the transaction.</td>
</tr>
<tr>
<td>19</td>
<td>Responder: Error was detected by the responder of the transaction.</td>
</tr>
<tr>
<td>20</td>
<td>Requester: Error was detected by the requester of the transaction.</td>
</tr>
<tr>
<td>21</td>
<td>First Error: If multiple errors are logged for a section type, this is the first error in the chronological sequence. Setting of this bit is optional.</td>
</tr>
<tr>
<td>22</td>
<td>Overflow: Additional errors occurred and were not logged due to lack of logging resources.</td>
</tr>
<tr>
<td>63:23</td>
<td>Reserved.</td>
</tr>
</tbody>
</table>

#### Table N-4 Error Types

<table>
<thead>
<tr>
<th>Encoding</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>ERR_INTERNAL Error detected internal to the component.</td>
</tr>
<tr>
<td>16</td>
<td>ERR_BUS Error detected in the bus.</td>
</tr>
</tbody>
</table>

**Detailed Internal Errors**
### Encoding | Description
--- | ---
4 | ERR_MEM Storage error in memory (DRAM).
5 | ERR_TLB Storage error in TLB.
6 | ERR_CACHE Storage error in cache.
7 | ERR_FUNCTION Error in one or more functional units.
8 | ERR_SELFTEST component failed self test.
9 | ERR_FLOW Overflow or undervalue of internal queue.

#### Detailed Bus Errors

17 | ERR_MAP Virtual address not found on IO-TLB or IO-PDIR.
18 | ERR_IMPROPER Improper access error.
19 | ERR_UNIMPL Access to a memory address which is not mapped to any component
20 | ERR_LOL Loss of Lockstep
21 | ERR_RESPONSE Response not associated with a request
22 | ERR_PARITY Bus parity error (must also set the A, C, or D Bits).
23 | ERR_PROTOCOL Detection of a protocol error.
24 | ERR_ERROR Detection of a PATH_ERROR
25 | ERR_TIMEOUT Bus operation timeout.
26 | ERR_POISONED A read was issued to data that has been poisoned.

All Others Reserved.

---

### N.2.2 Section Descriptor

#### Table N-5 Section Descriptor

<table>
<thead>
<tr>
<th>Mnemonic</th>
<th>Byte Offset</th>
<th>Byte Length</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Section Offset</td>
<td>0</td>
<td>4</td>
<td>Offset in bytes of the section body from the base of the record header.</td>
</tr>
<tr>
<td>Section Length</td>
<td>4</td>
<td>4</td>
<td>The length in bytes of the section body.</td>
</tr>
</tbody>
</table>
| Revision | 8 | 2 | This is a 2-byte field representing a major and minor version number for the error record definition in BCD format. The interpretation of the major and minor version number is as follows:  
  · Byte 0 – Minor (00): An increase in this revision indicates that changes to the headers and sections are backward compatible with software that uses earlier revisions. Addition of new GUID types, errata fixes or clarifications are covered by a bump up.  
  · Byte 1 – Major (01): An increase in this revision indicates that the changes are not backward compatible from a software perspective |
Validation Bits

- Bit 0: If 1, the FRUId field contains valid information
- Bit 1: If 1, the FRUString field contains valid information
- Bits 7:2: Reserved, must be zero.

Reserved

Flag field contains information that describes the error section as follows:

- Bit 0: Primary: If set, identifies the section as the section to be associated with the error condition. This allows for FRU determination and for error recovery operations. By identifying a primary section, the consumer of an error record can determine which section to focus on. It is not always possible to identify a primary section so this flag should be taken as a hint.
- Bit 1: Containment Warning: If set, the error was not contained within the processor or memory hierarchy and the error may have propagated to persistent storage or network.
- Bit 2: Reset: If set, the component has been reset and must be re-initialized or re-enabled by the operating system prior to use.
- Bit 3: Error threshold exceeded: If set, OS may choose to discontinue use of this resource.
- Bit 4: Resource not accessible: If set, the resource could not be queried for error information due to conflicts with other system software or resources. Some fields of the section will be invalid.
- Bit 5: Latent error: If set this flag indicates that action has been taken to ensure error containment (such as poisoning data), but the error has not been fully corrected and the data has not been consumed. System software may choose to take further corrective action before the data is consumed.
- Bit 6: Propagated: If set this flag indicates the section is to be associated with an error that has been propagated due to hardware poisoning. This implies the error is a symptom of another error. It is not always possible to ascertain whether this is the case for an error, therefore if the flag is not set, it is unknown whether the error was propagated. This helps determining FRU when dealing with HW failures.
- Bit 7: Overflow: If set this flag indicates the firmware has detected an overflow of buffers/queues that are used to accumulate, collect, or report errors (e.g. the error status control block exposed to the OS). When this occurs, some error records may be lost.

Bit 8 through 31: Reserved.
Section Type 16

This field holds a pre-assigned GUID value indicating that it is a section of a particular error. The different error section types are as defined below:

Processor Generic
- {0x9876CCAD, 0x4784, 0x4b3db, {0x86, 0x5E, 0x16, 0xF1, 0x93, 0xC4, 0xF3, 0xDB}}

Processor Specific
- IA32/X64: {0xDC3EA0B0, 0xA144, 0x4797, {0xB9, 0x5B, 0x53, 0xFA, 0x24, 0x2B, 0x6E, 0x1D}}
- IPF: {0xe429fa1, 0x3cb7, 0x11d4, {0xb, 0xca, 0x7, 0x00, 0x80, 0xc7, 0x3c, 0x88, 0x81}}
- ARM: {0xE19E3D16, 0x8C11, 0x11E4, {0x9C, 0xAA, 0xC2, 0x05, 0x1D, 0x5D, 0x46, 0x80}}

NOTE: In addition to the types listed above, there may exist vendor specific GUIDs that describe vendor specific section types.

Platform Memory
- {0xA5BC1114, 0x6F64, 0x4EDE, {0xB8, 0x63, 0x3E, 0x83, 0xED, 0x7C, 0x83, 0xB1}}

PCIe
- {0xD995E954, 0xBB1C, 0x430F, {0xAD, 0x91, 0xB4, 0x4D, 0xCB, 0x3C, 0x6F, 0x35}}

Firmware Error Record Reference
- {0xB1212A96, 0x09ED, 0x9496, {0x94, 0x71, 0x8D, 0x72, 0x9C, 0x8E, 0x69, 0xED}}

PCI/PCI-X Bus
- {0xC5753963, 0x3B84, 0x4095, {0xBF, 0x78, 0xED, 0xDA, 0xD3, 0x9F, 0xC9, 0xDD}}

PCI Component/Device
- {0xEB5E4685, 0xCA66, 0x4769, {0xB6, 0xA2, 0x26, 0x06, 0x8B, 0x00, 0x13, 0x26}}

DMAr Generic
- {0x5B51FEF7, 0x79D, 0x4434, {0x8F, 0x1B, 0xAA, 0x62, 0x3E, 0xC2, 0x64}}

Intel® VT for Directed I/O specific DMAr section
- {0x71761D37, 0x32B2, 0x45cd, {0xA7, 0xD0, 0xB0, 0xFE, 0xDD, 0x93, 0xEE, 0xCF}}

IOMMU specific DMAr section
- {0x036F84E1, 0x7F37, 0x428c, {0xA7, 0x9E, 0x57, 0x5F, 0xDF, 0xAA, 0x84, 0xEC}}

CXL Component Events
- See Table N-42 CXL Component Events Section

FRU Id 32

GUID representing the FRU ID, if it exists, for the section reporting the error. The default value is zero indicating an invalid FRU ID. System software can use this to uniquely identify a physical device for tracking purposes. Association of a GUID to a physical device is done by the platform in an implementation-specific way (i.e., PCIe Device can lock a GUID to a PCIe Device ID).
N.2.3 Non-standard Section Body

Information that does not conform to one the standard formats (i.e., those defined in sections 2.4 through 2.9 of this document) may be recorded in the error record in a non-standard section. The type (e.g. format) of a non-standard section is identified by the GUID populated in the Section Descriptor’s Section Type field. This allows the information to be decoded by consumers if the format is externally documented. Examples of information that might be placed in a non-standard section include the IPF raw SAL error record, Error information recorded in implementation-specific PCI configuration space, and IPMI error information recorded in an IPMI SEL.

N.2.4 Processor Error Sections

The processor error sections are divided into two different components as described below:

1. Processor Generic Error Section: This section holds information about processor errors in a generic form and will be common across all processor architectures. An example or error information provided is the generic information of cache, tlb, etc., errors.

2. Processor Specific Error Section: This section consists of error information, which is specific to a processor architecture. In addition, certain processor architecture state at the time of error may also be captured in this section. This section is unique to each processor architecture (Itanium Processor Family, IA32/X64, ARM).

N.2.4.1 Generic Processor Error Section

The Generic Processor Error Section describes processor reported hardware errors for logical processors in the system.

Section Type: {0x9876CCAD, 0x47B4, 0x4bdb, {0xB6, 0x5E, 0x16, 0xF1, 0x93, 0xC4, 0xF3, 0xDB}}

<table>
<thead>
<tr>
<th>Mnemonic</th>
<th>Byte Offset</th>
<th>Byte Length</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Section Severity</td>
<td>48</td>
<td>4</td>
<td>This field indicates the severity associated with the error section. 0 – Recoverable (also called non-fatal uncorrected) 1 – Fatal 2 – Corrected</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>3 – Informational  All other values are reserved. Note that severity of &quot;Informational&quot; indicates that the section contains extra information that can be safely ignored by error handling software.</td>
</tr>
<tr>
<td>FRU Text</td>
<td>52</td>
<td>20</td>
<td>ASCII string identifying the FRU hardware.</td>
</tr>
</tbody>
</table>
### Table N-6 Processor Generic Error Section

<table>
<thead>
<tr>
<th>Name</th>
<th>Byte Offset</th>
<th>Byte Length</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Validation Bits</td>
<td>0</td>
<td>8</td>
<td>The validation bit mask indicates whether or not each of the following fields is valid in this section. Bit 0 – Processor Type Valid Bit 1 – Processor ISA Valid Bit 2 – Processor Error Type Valid Bit 3 – Operation Valid Bit 4 – Flags Valid Bit 5 – Level Valid Bit 6 – CPU Version Valid Bit 7 – CPU Brand Info Valid Bit 8 – CPU Id Valid Bit 9 – Target Address Valid Bit 10 – Requester Identifier Valid Bit 11 – Responder Identifier Valid Bit 12 – Instruction IP Valid All other bits are reserved and must be zero.</td>
</tr>
<tr>
<td>Processor Type</td>
<td>8</td>
<td>1</td>
<td>Identifies the type of the processor architecture. 0: IA32/X64 1: IA64 2: ARM All other values reserved.</td>
</tr>
<tr>
<td>Processor ISA</td>
<td>9</td>
<td>1</td>
<td>Identifies the type of the instruction set executing when the error occurred: 0: IA32 1: IA64 2: X64 3: ARM A32/T32 4: ARM A64 All other values are reserved.</td>
</tr>
<tr>
<td>Processor Error Type</td>
<td>10</td>
<td>1</td>
<td>Indicates the type of error that occurred: 0x00: Unknown 0x01: Cache Error 0x02: TLB Error 0x04: Bus Error 0x08: Micro-Architectural Error All other values reserved.</td>
</tr>
<tr>
<td>Operation</td>
<td>11</td>
<td>1</td>
<td>Indicates the type of operation: 0: Unknown or generic 1: Data Read 2: Data Write 3: Instruction Execution All other values reserved.</td>
</tr>
</tbody>
</table>
### Flags

<table>
<thead>
<tr>
<th>Name</th>
<th>Byte Offset</th>
<th>Byte Length</th>
<th>Description</th>
</tr>
</thead>
</table>
| Flags         | 12          | 1           | Indicates additional information about the error:  
_bit 0: Restartable – If 1, program execution can be restarted reliably after the error.  
_bit 1: Precise IP – If 1, the instruction IP captured is directly associated with the error.  
_bit 2: Overflow – If 1, a machine check overflow occurred (a second error occurred while the results of a previous error were still in the error reporting resources).  
_bit 3: Corrected – If 1, the error was corrected by hardware and/or firmware.  
All other bits are reserved and must be zero. |

<table>
<thead>
<tr>
<th>Name</th>
<th>Byte Offset</th>
<th>Byte Length</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Level</td>
<td>13</td>
<td>1</td>
<td>Level of the structure where the error occurred, with 0 being the lowest level of cache.</td>
</tr>
<tr>
<td>Reserved</td>
<td>14</td>
<td>2</td>
<td>Must be zero.</td>
</tr>
</tbody>
</table>
| CPU Version Info | 16         | 8           | This field represents the CPU Version Information and returns Family, Model, and stepping information (e.g. As provided by CPUID instruction with EAX=1 input with output values from EAX on the IA32/X64 processor or as provided by CPUID Register 3 register – Version Information on IA64 processors). On ARM processors, this field will be provided as:  
Bits 127:64 - Reserved and must be zero  
Bits 63:0 - MIDR_EL1 of the processor |
| CPU Brand String | 24         | 128         | This field represents the null-terminated ASCII Processor Brand String (e.g. As provided by the CPUID instruction with EAX=0x80000002 and ECX=0x80000003 for IA32/X64 processors or the return from PAL_BRAND_INFO for IA64 processors). This field is optional for ARM processors. |
| Processor ID  | 152         | 8           | This value uniquely identifies the logical processor (e.g. As programmed into the local APIC ID register on IA32/X64 processors or programmed into the LID register on IA64 processors). On ARM processors, this field will be provided as programmed in the architected MPIDR_EL1. |
| Target Address | 160        | 8           | Identifies the target address associated with the error. |
| Requestor Identifier | 168     | 8           | Identifies the requestor associated with the error. |
| Responder Identifier | 176       | 8           | Identifies the responder associated with the error. |
| Instruction IP | 184         | 8           | Identifies the instruction pointer when the error occurred. |

## N.2.4.2 IA32/X64 Processor Error Section

Type:{0xDC3E0A0B0, 0xA144, 0x4797, {0xB9, 0x5B, 0x53, 0xFA, 0x24, 0x2B, 0x6E, 0x1D}}
**N.2.4.2.1 IA32/X64 Processor Error Information Structure**

As described above, the processor error section contains a collection of structures called Processor Error Information Structures that contain processor structure specific error information. This section details the layout of the Processor Error Information Structure and the detailed check information which is contained within.

### Table N-8 IA32/X64 Processor Error Information Structure

<table>
<thead>
<tr>
<th>Mnemonic</th>
<th>Byte Offset</th>
<th>Byte Length</th>
<th>Description</th>
</tr>
</thead>
</table>
| Error Structure Type      | 0           | 16          | This field holds a pre-assigned GUID indicating the type of Processor Error Information structure. The following Processor Error Information Structure Types have pre-defined GUID.  
  - Cache Error Information (Cache Check)  
  - TLB Error Information (TLB Check)  
  - Bus Error Information (Bus Check)  
  - Micro-architecture Specific Error Information (MS Check) |
### IA32/X64 Cache Check Structure

Type: {0xA55701F5, 0xE3EF, 0x43de, {0xAC, 0x72, 0x24, 0x9B, 0x57, 0x3F, 0xAD, 0x2C}}

#### Table N-9 IA32/X64 Cache Check Structure

<table>
<thead>
<tr>
<th>Field Name</th>
<th>Bits</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Validation Bits</td>
<td>15:0</td>
<td>Indicates which fields in the Cache Check structure are valid:</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Bit 0 – Transaction Type Valid</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Bit 1 – Operation Valid</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Bit 2 – Level Valid</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Bit 3 – Processor Context Corrupt Valid</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Bit 4 – Uncorrected Valid</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Bit 5 – Precise IP Valid</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Bit 6 – Restartable Valid</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Bit 7 – Overflow Valid</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Bits 8 – 15 Reserved</td>
</tr>
<tr>
<td>Transaction Type</td>
<td>17:16</td>
<td>Type of cache error:</td>
</tr>
<tr>
<td></td>
<td></td>
<td>0 – Instruction</td>
</tr>
<tr>
<td></td>
<td></td>
<td>1 – Data Access</td>
</tr>
<tr>
<td></td>
<td></td>
<td>2 – Generic</td>
</tr>
<tr>
<td></td>
<td></td>
<td>All other values are reserved</td>
</tr>
<tr>
<td>Operation</td>
<td>21:18</td>
<td>Type of cache operation that caused the error:</td>
</tr>
<tr>
<td></td>
<td></td>
<td>0 – generic error (type of error cannot be determined)</td>
</tr>
<tr>
<td></td>
<td></td>
<td>1 – generic read (type of instruction or data request cannot be determined)</td>
</tr>
<tr>
<td></td>
<td></td>
<td>2 – generic write (type of instruction or data request cannot be determined)</td>
</tr>
<tr>
<td></td>
<td></td>
<td>3 – data read</td>
</tr>
<tr>
<td></td>
<td></td>
<td>4 – data write</td>
</tr>
<tr>
<td></td>
<td></td>
<td>5 – instruction fetch</td>
</tr>
<tr>
<td></td>
<td></td>
<td>6 – prefetch</td>
</tr>
<tr>
<td></td>
<td></td>
<td>7 – eviction</td>
</tr>
<tr>
<td></td>
<td></td>
<td>8 – snoop</td>
</tr>
<tr>
<td></td>
<td></td>
<td>All other values are reserved</td>
</tr>
<tr>
<td>Level</td>
<td>24:22</td>
<td>Cache Level</td>
</tr>
</tbody>
</table>

**Validation Bits**
- Bit 0 – Check Info Valid
- Bit 1 – Target Address Identifier Valid
- Bit 2 – Requestor Identifier Valid
- Bit 3 – Responder Identifier Valid
- Bit 4 – Instruction Pointer Valid
- Bits 5-63 – Reserved

**Check Information**
- StructureErrorType specific error check structure.

**Target Identifier**
- Identifies the target associated with the error.

**Requestor Identifier**
- Identifies the requestor associated with the error.

**Responder Identifier**
- Identifies the responder associated with the error.

**Instruction Pointer**
- Identifies the instruction executing when the error occurred.
IA32/X64 TLB Check Structure
Type:{0xFC06B535, 0x5E1F, 0x4562, {0x9F, 0x25, 0x0A, 0x3B, 0x9A, 0xDB, 0x63, 0xC3}}

Table N-10 IA32/X64 TLB Check Structure

<table>
<thead>
<tr>
<th>Field Name</th>
<th>Bits</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Processor Context Corrupt</td>
<td>25</td>
<td>This field indicates that the processor context might have been corrupted.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>0 - Processor context not corrupted</td>
</tr>
<tr>
<td></td>
<td></td>
<td>1 - Processor context corrupted</td>
</tr>
<tr>
<td>Uncorrected</td>
<td>26</td>
<td>This field indicates whether the error was corrected or uncorrected:</td>
</tr>
<tr>
<td></td>
<td></td>
<td>0: Corrected</td>
</tr>
<tr>
<td></td>
<td></td>
<td>1: Uncorrected</td>
</tr>
<tr>
<td>Precise IP</td>
<td>27</td>
<td>This field indicates that the instruction pointer pushed onto the stack is</td>
</tr>
<tr>
<td></td>
<td></td>
<td>directly associated with the error</td>
</tr>
<tr>
<td>Restartable IP</td>
<td>28</td>
<td>This field indicates that program execution can be restarted reliably</td>
</tr>
<tr>
<td></td>
<td></td>
<td>at the instruction pointer pushed onto the stack</td>
</tr>
<tr>
<td>Overflow</td>
<td>29</td>
<td>This field indicates an error overflow occurred</td>
</tr>
<tr>
<td></td>
<td></td>
<td>0 - Overflow not occurred</td>
</tr>
<tr>
<td></td>
<td></td>
<td>1 - Overflow occurred</td>
</tr>
<tr>
<td></td>
<td>63:30</td>
<td>Reserved</td>
</tr>
</tbody>
</table>

Field Name Bits Description

- Validation Bits 15:0 Indicate which fields in the Cache_Check structure are valid
  - Bit 0 – Transaction Type Valid
  - Bit 1 – Operation Valid
  - Bit 2 – Level Valid
  - Bit 3 – Processor Context Corrupt Valid
  - Bit 4 – Uncorrected Valid
  - Bit 5 – Precise IP Valid
  - Bit 6 – Restartable IP Valid
  - Bit 7 – Overflow Valid
  - Bit 8 – 15 Reserved

- Transaction Type 17:16 Type of TLB error
  - 0 – Instruction
  - 1 – Data Access
  - 2 – Generic
  - All other values are reserved

- Operation 21:18 Type of TLB access operation that caused the machine check:
  - 0 – generic error (type of error cannot be determined)
  - 1 – generic read (type of instruction or data request cannot be determined)
  - 2 – generic write (type of instruction or data request cannot be determined)
  - 3 – data read
  - 4 – data write
  - 5 – instruction fetch
  - 6 – prefetch
  - All other values are reserved.
IA32/X64 Bus Check Structure

Type:{0x1CF3F8B3, 0xC5B1, 0x49a2, {0xAA, 0x59, 0x5E, 0xEF, 0x92, 0xFF, 0xA6, 0x3C}}

Table N-11 IA32/X64 Bus Check Structure

<table>
<thead>
<tr>
<th>Field Name</th>
<th>Bits</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Validation Bits</td>
<td>15:0</td>
<td>Indicate which fields in the Bus_Check structure are valid</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Bit 0 – Transaction Type Valid</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Bit 1 – Operation Valid</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Bit 2 – Level Valid</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Bit 3 – Processor Context Corrupt Valid</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Bit 4 – Uncorrected Valid</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Bit 5 – Precise IP Valid</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Bit 6 – Restartable IP Valid</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Bit 7 – Overflow Valid</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Bit 8 – Participation Type Valid</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Bit 9 – Time Out Valid</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Bit 10 – Address Space Valid</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Bit 11 – 15 Reserved</td>
</tr>
<tr>
<td>Transaction Type</td>
<td>17:16</td>
<td>Type of Bus error</td>
</tr>
<tr>
<td></td>
<td></td>
<td>0 – Instruction</td>
</tr>
<tr>
<td></td>
<td></td>
<td>1 – Data Access</td>
</tr>
<tr>
<td></td>
<td></td>
<td>2 – Generic</td>
</tr>
<tr>
<td></td>
<td></td>
<td>All other values are reserved</td>
</tr>
</tbody>
</table>
### IA32/X64 MS Check Field Description

**Type:** \{0x48AB7F57, 0xDC34, 0x4f6c, \{0xA7, 0xD3, 0xB0, 0xB5, 0xB0, 0xA7, 0x43, 0x14\}\}

<table>
<thead>
<tr>
<th>Field Name</th>
<th>Bits</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Operation</td>
<td>21:18</td>
<td>Type of bus access operation that caused the machine check:</td>
</tr>
<tr>
<td></td>
<td></td>
<td>0 – generic error (type of error cannot be determined)</td>
</tr>
<tr>
<td></td>
<td></td>
<td>1 – generic read (type of instruction or data request cannot be determined)</td>
</tr>
<tr>
<td></td>
<td></td>
<td>2 – generic write (type of instruction or data request cannot be determined)</td>
</tr>
<tr>
<td></td>
<td></td>
<td>3 – data read</td>
</tr>
<tr>
<td></td>
<td></td>
<td>4 – data write</td>
</tr>
<tr>
<td></td>
<td></td>
<td>5 – instruction fetch</td>
</tr>
<tr>
<td></td>
<td></td>
<td>6 – prefetch</td>
</tr>
<tr>
<td></td>
<td></td>
<td>All other values are reserved.</td>
</tr>
<tr>
<td>Level</td>
<td>24:22</td>
<td>Indicate which level of the bus hierarchy the error occurred in.</td>
</tr>
<tr>
<td>Processor Context Corrupt</td>
<td>25</td>
<td>This field indicates that the processor context might have been corrupted.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>0 - Processor context not corrupted</td>
</tr>
<tr>
<td></td>
<td></td>
<td>1 - Processor context corrupted</td>
</tr>
<tr>
<td>Uncorrected</td>
<td>26</td>
<td>This field indicates whether the error was corrected or uncorrected:</td>
</tr>
<tr>
<td></td>
<td></td>
<td>0: Corrected</td>
</tr>
<tr>
<td></td>
<td></td>
<td>1: Uncorrected</td>
</tr>
<tr>
<td>PreciseIP</td>
<td>27</td>
<td>This field indicates that the instruction pointer pushed onto the stack is directly associated with the error.</td>
</tr>
<tr>
<td>Restartable IP</td>
<td>28</td>
<td>This field indicates the program execution can be restarted reliably at the instruction pointer pushed onto the stack.</td>
</tr>
<tr>
<td>Overflow</td>
<td>29</td>
<td>This field indicates an error overflow occurred</td>
</tr>
<tr>
<td></td>
<td></td>
<td>0 - Overflow not occurred</td>
</tr>
<tr>
<td></td>
<td></td>
<td>1 - Overflow occurred</td>
</tr>
<tr>
<td>Participation Type</td>
<td>31:30</td>
<td>Type of Participation</td>
</tr>
<tr>
<td></td>
<td></td>
<td>0 – Local Processor originated request</td>
</tr>
<tr>
<td></td>
<td></td>
<td>1 – Local processor Responded to request</td>
</tr>
<tr>
<td></td>
<td></td>
<td>2 – Local processor Observed</td>
</tr>
<tr>
<td></td>
<td></td>
<td>3 – Generic</td>
</tr>
<tr>
<td>Time Out</td>
<td>32</td>
<td>This field indicates that the request timed out.</td>
</tr>
<tr>
<td>Address Space</td>
<td>34:33</td>
<td>0 – Memory Access</td>
</tr>
<tr>
<td></td>
<td></td>
<td>1 – Reserved</td>
</tr>
<tr>
<td></td>
<td></td>
<td>2 – I/O</td>
</tr>
<tr>
<td></td>
<td></td>
<td>3 – Other Transaction</td>
</tr>
<tr>
<td></td>
<td>63:35</td>
<td>Reserved</td>
</tr>
</tbody>
</table>
Table N-12 IA32/X64 MS Check Field Description

<table>
<thead>
<tr>
<th>Field Name</th>
<th>Bits</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Validation Bits</td>
<td>15:0</td>
<td>Indicate which fields in the Cache_Check structure are valid</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Bit 0 – Error Type Valid</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Bit 1 – Processor Context Corrupt Valid</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Bit 2 – Uncorrected Valid</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Bit 3 – Precise IP Valid</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Bit 4 – Restartable IP Valid</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Bit 5 – Overflow Valid</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Bit 6 – 15 Reserved</td>
</tr>
<tr>
<td>Error Type</td>
<td>18:16</td>
<td>Identifies the operation that caused the error:</td>
</tr>
<tr>
<td></td>
<td></td>
<td>0 – No Error</td>
</tr>
<tr>
<td></td>
<td></td>
<td>1 – Unclassified</td>
</tr>
<tr>
<td></td>
<td></td>
<td>2 – Microcode ROM Parity Error</td>
</tr>
<tr>
<td></td>
<td></td>
<td>3 – External Error</td>
</tr>
<tr>
<td></td>
<td></td>
<td>4 – FRC Error</td>
</tr>
<tr>
<td></td>
<td></td>
<td>5 – Internal Unclassified</td>
</tr>
<tr>
<td></td>
<td></td>
<td>All other value are processor specific.</td>
</tr>
<tr>
<td>Processor Context</td>
<td>19</td>
<td>This field indicates that the processor context might have been corrupted.</td>
</tr>
<tr>
<td>Corrupt</td>
<td></td>
<td>0 - Processor context not corrupted</td>
</tr>
<tr>
<td></td>
<td></td>
<td>1 - Processor context corrupted</td>
</tr>
<tr>
<td>Uncorrected</td>
<td>20</td>
<td>This field indicates whether the error was corrected or uncorrected:</td>
</tr>
<tr>
<td></td>
<td></td>
<td>0: Correct</td>
</tr>
<tr>
<td></td>
<td></td>
<td>1: Uncorrected</td>
</tr>
<tr>
<td>Precise IP</td>
<td>21</td>
<td>This field indicates that the instruction pointer pushed onto the stack is</td>
</tr>
<tr>
<td></td>
<td></td>
<td>directly associated with the error.</td>
</tr>
<tr>
<td>Restartable IP</td>
<td>22</td>
<td>This field indicates the program execution can be restarted reliably at the</td>
</tr>
<tr>
<td></td>
<td></td>
<td>instruction pointer pushed onto the stack.</td>
</tr>
<tr>
<td>Overflow</td>
<td>23</td>
<td>This field indicates an error overflow occurred</td>
</tr>
<tr>
<td></td>
<td></td>
<td>0 - Overflow not occurred</td>
</tr>
<tr>
<td></td>
<td></td>
<td>1 - Overflow occurred</td>
</tr>
<tr>
<td></td>
<td>63:24</td>
<td>Reserved</td>
</tr>
</tbody>
</table>

N.2.4.2.2 IA32/X64 Processor Context Information Structure

As described above, the processor error section contains a collection of structures called Processor Context Information that contain processor context state specific to the IA32/X64 processor architecture. This section details the layout of the Processor Context Information Structure and the detailed processor context type information.

Table N-13 IA32/X64 Processor Context Information

<table>
<thead>
<tr>
<th>Mnemonic</th>
<th>Byte Offset</th>
<th>Byte Length</th>
<th>Description</th>
</tr>
</thead>
</table>


# Table N-14 IA32 Register State

<table>
<thead>
<tr>
<th>Offset</th>
<th>Length</th>
<th>Field</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>4 bytes</td>
<td>EAX</td>
</tr>
<tr>
<td>4</td>
<td>4 bytes</td>
<td>EBX</td>
</tr>
<tr>
<td>8</td>
<td>4 bytes</td>
<td>ECX</td>
</tr>
<tr>
<td>12</td>
<td>4 bytes</td>
<td>EDX</td>
</tr>
<tr>
<td>16</td>
<td>4 bytes</td>
<td>ESI</td>
</tr>
<tr>
<td>20</td>
<td>4 bytes</td>
<td>EDI</td>
</tr>
<tr>
<td>24</td>
<td>4 bytes</td>
<td>EBP</td>
</tr>
<tr>
<td>28</td>
<td>4 bytes</td>
<td>ESP</td>
</tr>
<tr>
<td>32</td>
<td>2 bytes</td>
<td>CS</td>
</tr>
<tr>
<td>34</td>
<td>2 bytes</td>
<td>DS</td>
</tr>
<tr>
<td>36</td>
<td>2 bytes</td>
<td>SS</td>
</tr>
<tr>
<td>38</td>
<td>2 bytes</td>
<td>ES</td>
</tr>
<tr>
<td>40</td>
<td>2 bytes</td>
<td>FS</td>
</tr>
<tr>
<td>42</td>
<td>2 bytes</td>
<td>GS</td>
</tr>
<tr>
<td>44</td>
<td>4 bytes</td>
<td>EFLAGS</td>
</tr>
<tr>
<td>48</td>
<td>4 bytes</td>
<td>EIP</td>
</tr>
<tr>
<td>52</td>
<td>4 bytes</td>
<td>CR0</td>
</tr>
<tr>
<td>56</td>
<td>4 bytes</td>
<td>CR1</td>
</tr>
<tr>
<td>60</td>
<td>4 bytes</td>
<td>CR2</td>
</tr>
<tr>
<td>64</td>
<td>4 bytes</td>
<td>CR3</td>
</tr>
<tr>
<td>68</td>
<td>4 bytes</td>
<td>CR4</td>
</tr>
</tbody>
</table>

Table N-14 shows the register context type 2, 32-bit mode execution context.
Table N-15 shows the register context type 3, 64-bit mode execution context.

### Table N-15 X64 Register State

<table>
<thead>
<tr>
<th>Offset</th>
<th>Length</th>
<th>Field</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>8 bytes</td>
<td>RAX</td>
</tr>
<tr>
<td>8</td>
<td>8 bytes</td>
<td>RBX</td>
</tr>
<tr>
<td>16</td>
<td>8 bytes</td>
<td>RCX</td>
</tr>
<tr>
<td>24</td>
<td>8 bytes</td>
<td>RDX</td>
</tr>
<tr>
<td>32</td>
<td>8 bytes</td>
<td>RSI</td>
</tr>
<tr>
<td>40</td>
<td>8 bytes</td>
<td>RDI</td>
</tr>
<tr>
<td>48</td>
<td>8 bytes</td>
<td>RBP</td>
</tr>
<tr>
<td>56</td>
<td>8 bytes</td>
<td>RSP</td>
</tr>
<tr>
<td>64</td>
<td>8 bytes</td>
<td>R8</td>
</tr>
<tr>
<td>72</td>
<td>8 bytes</td>
<td>R9</td>
</tr>
<tr>
<td>80</td>
<td>8 bytes</td>
<td>R10</td>
</tr>
<tr>
<td>88</td>
<td>8 bytes</td>
<td>R11</td>
</tr>
<tr>
<td>96</td>
<td>8 bytes</td>
<td>R12</td>
</tr>
<tr>
<td>104</td>
<td>8 bytes</td>
<td>R13</td>
</tr>
<tr>
<td>112</td>
<td>8 bytes</td>
<td>R14</td>
</tr>
<tr>
<td>120</td>
<td>8 bytes</td>
<td>R15</td>
</tr>
<tr>
<td>128</td>
<td>2 bytes</td>
<td>CS</td>
</tr>
<tr>
<td>130</td>
<td>2 bytes</td>
<td>DS</td>
</tr>
<tr>
<td>132</td>
<td>2 bytes</td>
<td>SS</td>
</tr>
<tr>
<td>134</td>
<td>2 bytes</td>
<td>ES</td>
</tr>
<tr>
<td>136</td>
<td>2 bytes</td>
<td>FS</td>
</tr>
<tr>
<td>138</td>
<td>2 bytes</td>
<td>GS</td>
</tr>
<tr>
<td>140</td>
<td>4 bytes</td>
<td>Reserved</td>
</tr>
<tr>
<td>144</td>
<td>8 bytes</td>
<td>RFLAGS</td>
</tr>
<tr>
<td>152</td>
<td>8 bytes</td>
<td>EIP</td>
</tr>
<tr>
<td>160</td>
<td>8 bytes</td>
<td>CR0</td>
</tr>
<tr>
<td>168</td>
<td>8 bytes</td>
<td>CR1</td>
</tr>
<tr>
<td>176</td>
<td>8 bytes</td>
<td>CR2</td>
</tr>
<tr>
<td>184</td>
<td>8 bytes</td>
<td>CR3</td>
</tr>
<tr>
<td>192</td>
<td>8 bytes</td>
<td>CR4</td>
</tr>
</tbody>
</table>
N.2.4.3 IA64 Processor Error Section

Refer to the Intel Itanium Processor Family System Abstraction Layer specification for finding the IA64 specific error section body definition.

N.2.4.4 ARM Processor Error Section

Type: {0xE19E3D16, 0xBC11, 0x11E4, {0x9C, 0xAA, 0xC2, 0x05, 0x1D, 0x5D, 0x46, 0xB0}}

The ARM Processor Error Section may contain multiple instances of error information structures associated to a single error event. An error may propagate to other hardware components (e.g. poisoned data) or cause subsequent errors, all of which may be captured in a single ARM processor error section. The processor context information describes the observed state of the processor at the point of error detection.

It is optional for vendors to capture processor context information. The specifics of capturing processor context is vendor specific. Vendors must take care when handling errors that have originated whilst a processor was executing in a secure exception level. In those cases providing processor context information to non-secure agents could be unsafe and lead to security attacks.

Table N-16 ARM Processor Error Section

<table>
<thead>
<tr>
<th>Mnemonic</th>
<th>Byte Offset</th>
<th>Byte Length</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Validation Bit</td>
<td>0</td>
<td>4</td>
<td>The validation bit mask indicates whether or not each of the following fields is valid in this section. Bit 0 – MPIDR Valid Bit 1 – Error affinity level Valid Bit 2 - Running State Bit 3 – Vendor Specific Info Valid All other bits are reserved and must be zero.</td>
</tr>
<tr>
<td>ERR_INFO_- NUM</td>
<td>4</td>
<td>2</td>
<td>ERR_INFO_NUM is the number of Processor Error Information Structures (must be 1 or greater)</td>
</tr>
<tr>
<td>CONTEXT_- INFO_NUM</td>
<td>6</td>
<td>2</td>
<td>CONTEXT_INFO_NUM is the number of Context Information Structures</td>
</tr>
<tr>
<td>Section Length</td>
<td>8</td>
<td>4</td>
<td>This describes the total size of the ARM processor error section</td>
</tr>
</tbody>
</table>
### Error affinity level

<table>
<thead>
<tr>
<th>Mnemonic</th>
<th>Byte Offset</th>
<th>Byte Length</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Error affinity level</td>
<td>12</td>
<td>1</td>
<td>For errors that can be attributed to a specific affinity level, this field defines the affinity level at which the error was produced, detected, and/or consumed. This is a value between 0 and 3. All other values (4-255) are reserved. For example, a vendor may choose to define affinity levels as follows: Level 0: errors that can be precisely attributed to a specific CPU (e.g. due to a synchronous external abort) Level 1: Cache parity and/or ECC errors detected at cache of affinity level 1 (e.g. only attributed to higher level cache due to prefetching and/or error propagation) <strong>NOTE:</strong> Detailed meanings and groupings of affinity level are chip and/or platform specific. The affinity level described here must be consistent with the platform definitions used MPIDR. For cache/TLB errors, the cache/TLB level is provided by the cache/TLB error structure, which may differ from affinity level.</td>
</tr>
</tbody>
</table>

### Reserved

<table>
<thead>
<tr>
<th>Mnemonic</th>
<th>Byte Offset</th>
<th>Byte Length</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Reserved</td>
<td>13</td>
<td>3</td>
<td>Must be zero</td>
</tr>
</tbody>
</table>

### MPIDR_EL1

<table>
<thead>
<tr>
<th>Mnemonic</th>
<th>Byte Offset</th>
<th>Byte Length</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>MPIDR_EL1</td>
<td>16</td>
<td>8</td>
<td>This field is valid for “attributable errors” that can be attributed to a specific CPU, cache, or cluster. This is the processor’s unique ID in the system.</td>
</tr>
</tbody>
</table>

### MIDR_EL1

<table>
<thead>
<tr>
<th>Mnemonic</th>
<th>Byte Offset</th>
<th>Byte Length</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>MIDR_EL1</td>
<td>24</td>
<td>8</td>
<td>This field provides identification information of the chip, including an implementer code for the device and a device ID number</td>
</tr>
</tbody>
</table>

### Running State

<table>
<thead>
<tr>
<th>Mnemonic</th>
<th>Byte Offset</th>
<th>Byte Length</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Running State</td>
<td>32</td>
<td>4</td>
<td>Bit 0 – Processor running. If this bit is set, “PSCI State” field must be zero. All other bits are reserved and must be zero.</td>
</tr>
</tbody>
</table>

### PSCI State

<table>
<thead>
<tr>
<th>Mnemonic</th>
<th>Byte Offset</th>
<th>Byte Length</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>PSCI State</td>
<td>36</td>
<td>4</td>
<td>This field provides PSCI state of the processor, as defined in ARM PSCI document. This field is valid when bit 32 of “Running State” field is zero.</td>
</tr>
</tbody>
</table>

### Processor Error Information Structure

<table>
<thead>
<tr>
<th>Mnemonic</th>
<th>Byte Offset</th>
<th>Byte Length</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Processor Error Information Structure</td>
<td>40</td>
<td>Nx32</td>
<td>This is a variable-length structure consisting of N different 32 byte structures per Table N-17, each representing a single processor error information structure. The value of N ranges from 1-255 and is as indicated by ERR_INFO_NUM field in this table.</td>
</tr>
</tbody>
</table>

### Processor Context

<table>
<thead>
<tr>
<th>Mnemonic</th>
<th>Byte Offset</th>
<th>Byte Length</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Processor Context</td>
<td>40 + Nx32</td>
<td>MxP</td>
<td>This is a variable size field consisting of M different P byte structures providing the information for the processor context state such as general purpose registers (GPRs) and special purpose registers (SPRs) as defined in Table 266 or 267 (depending on the context type). The value of M ranges from 0-65536 and is indicated by the CONTEXT_INFO_NUM field in this table. Each processor context information structure is padded with zeros if the size is not a multiple of 16 bytes. The value of P is a variable length defined by the processor context structure per Table 266 and 267.</td>
</tr>
</tbody>
</table>

### Vendor Specific Error Info

<table>
<thead>
<tr>
<th>Mnemonic</th>
<th>Byte Offset</th>
<th>Byte Length</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Vendor Specific Error Info</td>
<td>40 + Nx32   + MxP</td>
<td>vendor specific</td>
<td>This is an optional variable field provided by vendors that prefer to provide additional details.</td>
</tr>
</tbody>
</table>
N.2.4.4.1 ARM Processor Error Information

As described above, the processor error section contains a collection of Processor Error Information structures that contain processor specific error information. This section details the layout of the Processor Error Information structure and the detailed information which is contained within.

Table N-17 ARM Processor Error Information Structure

<table>
<thead>
<tr>
<th>Mnemonic</th>
<th>Byte Offset</th>
<th>Byte Length</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Version</td>
<td>0</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>(revision of this table)</td>
</tr>
<tr>
<td>Length</td>
<td>1</td>
<td>1</td>
<td>32</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>(length in bytes)</td>
</tr>
<tr>
<td>Validation Bit</td>
<td>2</td>
<td>2</td>
<td>The validation bit mask indicates whether or not each of the following fields is valid in this section. Bit 0 – Multiple Error (Error Count) Valid Bit 1 – Flags Valid Bit 2 – Error Information Valid Bit 3 – Virtual Fault Address Bit 4 – Physical Fault Address All other bits are reserved and must be zero.</td>
</tr>
<tr>
<td>Type</td>
<td>4</td>
<td>1</td>
<td>Cache Error TLB Error Bus Error Micro-architectural Error All other values are reserved</td>
</tr>
<tr>
<td>Multiple Error (Error Count)</td>
<td>5</td>
<td>2</td>
<td>This field indicates whether multiple errors have occurred. In the case of multiple error with a valid count, this field will specify the error count. The value of this field is defined as follows: 0: Single Error 1: Multiple Errors 2-65535: Error Count (if known)</td>
</tr>
<tr>
<td>Flags</td>
<td>7</td>
<td>1</td>
<td>This field indicates flags that describe the error attributes. The value of this field is defined as follows: Bit 0 – First error captured Bit 1 – Last error captured Bit 2 – Propagated Bit 3 – Overflow All other bits are reserved and must be zero Note: Overflow bit indicates that firmware/hardware error buffers had experience an overflow, and it is possible that some error information has been lost.</td>
</tr>
<tr>
<td>Error Information</td>
<td>8</td>
<td>8</td>
<td>The error information structure is specific to each error type (described in tables below)</td>
</tr>
<tr>
<td>Virtual Fault Address</td>
<td>16</td>
<td>8</td>
<td>If known, this field indicates a virtual fault address associated with the error (e.g. when an error occurs in virtually indexed cache)</td>
</tr>
</tbody>
</table>
Table N-18 ARM Cache Error Structure

<table>
<thead>
<tr>
<th>Name</th>
<th>Bits</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Validation Bit</td>
<td>15:0</td>
<td>Indicates which fields in the Cache Check structure are valid: Bit 0 – Transaction Type Valid Bit 1 – Operation Valid Bit 2 – Level Valid Bit 3 – Processor Context Corrupt Valid Bit 4 – Corrected Valid Bit 5 – Precise PC Valid Bit 6 – Restartable PC Valid All other bits are reserved and must be zero.</td>
</tr>
<tr>
<td>Transaction Type</td>
<td>17:16</td>
<td>Type of cache error: 0 – Instruction 1 – Data Access 2 – Generic All other values are reserved.</td>
</tr>
<tr>
<td>Operation</td>
<td>21:18</td>
<td>Type of cache operation that caused the error: 0 – generic error (type of error cannot be determined) 1 – generic read (type of instruction or data request cannot be determined) 2 – generic write (type of instruction or data request cannot be determined) 3 – data read 4 – data write 5 – instruction fetch 6 – prefetch 7 – eviction 8 – snooping (the processor described in this record initiated a cache snoop that resulted in an error) 9 – snooped (The processor described in this record raised a cache error caused by another processor or device snooping into its cache) 10 – management All other values are reserved.</td>
</tr>
<tr>
<td>Level</td>
<td>24:22</td>
<td>Cache level</td>
</tr>
<tr>
<td>Processor Context Corrupt</td>
<td>25</td>
<td>This field indicates that the processor context might have been corrupted. 0 - Processor context not corrupted 1 - Processor context corrupted</td>
</tr>
<tr>
<td>Corrected</td>
<td>26</td>
<td>This field indicates whether the error was corrected or uncorrected: 1: Corrected 0: Uncorrected</td>
</tr>
<tr>
<td>Precise PC</td>
<td>27</td>
<td>This field indicates that the program counter that is directly associated with the error</td>
</tr>
<tr>
<td>Restartable PC</td>
<td>28</td>
<td>This field indicates that program execution can be restarted reliably at the PC associated with the error.</td>
</tr>
</tbody>
</table>
## Table N-19  ARM TLB Error Structure

<table>
<thead>
<tr>
<th>Name</th>
<th>Bits</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Reserved</td>
<td>63:29</td>
<td>Must be zero</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Name</th>
<th>Bits</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Validation Bit</td>
<td>15:0</td>
<td>Indicates which fields in the TLB error structure are valid:</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Bit 0 – Transaction Type Valid</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Bit 1 – Operation Valid</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Bit 2 – Level Valid</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Bit 3 – Processor Context Corrupt Valid</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Bit 4 – Corrected Valid</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Bit 5 – Precise PC Valid</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Bit 6 – Restartable PC Valid</td>
</tr>
<tr>
<td></td>
<td></td>
<td>All other bits are reserved and must be zero.</td>
</tr>
<tr>
<td>Transaction Type</td>
<td>17:16</td>
<td>Type of TLB error:</td>
</tr>
<tr>
<td></td>
<td></td>
<td>0 – Instruction</td>
</tr>
<tr>
<td></td>
<td></td>
<td>1 – Data Access</td>
</tr>
<tr>
<td></td>
<td></td>
<td>2 – Generic</td>
</tr>
<tr>
<td></td>
<td></td>
<td>All other values are reserved.</td>
</tr>
<tr>
<td>Operation</td>
<td>21:18</td>
<td>Type of TLB operation that caused the error:</td>
</tr>
<tr>
<td></td>
<td></td>
<td>0 – generic error (type of error cannot be determined)</td>
</tr>
<tr>
<td></td>
<td></td>
<td>1 – generic read (type of instruction or data request cannot be determined)</td>
</tr>
<tr>
<td></td>
<td></td>
<td>2 – generic write (type of instruction or data request cannot be determined)</td>
</tr>
<tr>
<td></td>
<td></td>
<td>3 – data read</td>
</tr>
<tr>
<td></td>
<td></td>
<td>4 – data write</td>
</tr>
<tr>
<td></td>
<td></td>
<td>5 – instruction fetch</td>
</tr>
<tr>
<td></td>
<td></td>
<td>6 – prefetch</td>
</tr>
<tr>
<td></td>
<td></td>
<td>7 – local management operation (the processor described in this record</td>
</tr>
<tr>
<td></td>
<td></td>
<td>initiated a TLB management operation that resulted in an error)</td>
</tr>
<tr>
<td></td>
<td></td>
<td>8 – external management operation (the processor described in this record</td>
</tr>
<tr>
<td></td>
<td></td>
<td>raised a TLB error caused by another processor or device broadcasting TLB</td>
</tr>
<tr>
<td></td>
<td></td>
<td>operations)</td>
</tr>
<tr>
<td></td>
<td></td>
<td>All other values are reserved.</td>
</tr>
<tr>
<td>Level</td>
<td>24:22</td>
<td>TLB level</td>
</tr>
<tr>
<td>Processor Context</td>
<td>25</td>
<td>This field indicates that the processor context might have been corrupted.</td>
</tr>
<tr>
<td>Context Corrupt</td>
<td></td>
<td>0 - Processor context not corrupted</td>
</tr>
<tr>
<td>Corrected</td>
<td>26</td>
<td>This field indicates whether the error was corrected or uncorrected:</td>
</tr>
<tr>
<td></td>
<td></td>
<td>1: Corrected</td>
</tr>
<tr>
<td></td>
<td></td>
<td>0: Uncorrected</td>
</tr>
<tr>
<td>Precise PC</td>
<td>27</td>
<td>This field indicates that the program counter that is directly associated</td>
</tr>
<tr>
<td></td>
<td></td>
<td>with the error</td>
</tr>
<tr>
<td>Restartable PC</td>
<td>28</td>
<td>This field indicates that program execution can be restarted reliably at</td>
</tr>
<tr>
<td></td>
<td></td>
<td>the PC associated with the error.</td>
</tr>
<tr>
<td>Reserved</td>
<td>63:29</td>
<td>Must be zero.</td>
</tr>
</tbody>
</table>
**Table N-20 ARM Bus Error Structure**

<table>
<thead>
<tr>
<th>Name</th>
<th>Bits</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Validation Bit</td>
<td>15:0</td>
<td>Indicates which fields in the Bus error structure are valid:</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Bit 0 – Transaction Type Valid</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Bit 1 – Operation Valid</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Bit 2 – Level Valid</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Bit 3 – Processor Context Corrupt Valid</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Bit 4 – Corrected Valid</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Bit 5 – Precise PC Valid</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Bit 6 – Restartable PC Valid</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Bit 7 – Participation Type Valid</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Bit 8 – Time Out Valid</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Bit 9 – Address Space Valid</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Bit 10 – Memory Attributes Valid</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Bit 11 – Access Mode valid</td>
</tr>
<tr>
<td></td>
<td></td>
<td>All other bits are reserved and must be zero.</td>
</tr>
<tr>
<td>Transaction Type</td>
<td>17:16</td>
<td>Type of bus error:</td>
</tr>
<tr>
<td></td>
<td></td>
<td>0 – Instruction</td>
</tr>
<tr>
<td></td>
<td></td>
<td>1 – Data Access</td>
</tr>
<tr>
<td></td>
<td></td>
<td>2 – Generic</td>
</tr>
<tr>
<td></td>
<td></td>
<td>All other values are reserved</td>
</tr>
<tr>
<td>Operation</td>
<td>21:18</td>
<td>Type of bus operation that caused the error:</td>
</tr>
<tr>
<td></td>
<td></td>
<td>0 – generic error (type of error cannot be determined)</td>
</tr>
<tr>
<td></td>
<td></td>
<td>1 – generic read (type of instruction or data request cannot be determined)</td>
</tr>
<tr>
<td></td>
<td></td>
<td>2 – generic write (type of instruction or data request cannot be determined)</td>
</tr>
<tr>
<td></td>
<td></td>
<td>3 – data read</td>
</tr>
<tr>
<td></td>
<td></td>
<td>4 – data write</td>
</tr>
<tr>
<td></td>
<td></td>
<td>5 – instruction fetch</td>
</tr>
<tr>
<td></td>
<td></td>
<td>6 – prefetch</td>
</tr>
<tr>
<td></td>
<td></td>
<td>All other values are reserved</td>
</tr>
<tr>
<td>Level</td>
<td>24:22</td>
<td>Affinity level at which the bus error occurred</td>
</tr>
<tr>
<td>Processor Context</td>
<td>25</td>
<td>This field indicates that the processor context might have been corrupted.</td>
</tr>
<tr>
<td>Corrupt</td>
<td></td>
<td>0 - Processor context not corrupted</td>
</tr>
<tr>
<td></td>
<td></td>
<td>1 - Processor context corrupted</td>
</tr>
<tr>
<td>Corrected</td>
<td>26</td>
<td>This field indicates whether the error was corrected or uncorrected:</td>
</tr>
<tr>
<td></td>
<td></td>
<td>1: Corrected</td>
</tr>
<tr>
<td></td>
<td></td>
<td>0: Uncorrected</td>
</tr>
<tr>
<td>Precise PC</td>
<td>27</td>
<td>This field indicates that the program counter that is directly associated</td>
</tr>
<tr>
<td></td>
<td></td>
<td>with the error</td>
</tr>
<tr>
<td>Restartable PC</td>
<td>28</td>
<td>This field indicates that program execution can be restarted reliably at</td>
</tr>
<tr>
<td></td>
<td></td>
<td>the PC associated with the error</td>
</tr>
</tbody>
</table>
ARM Vendor Specific Micro-Architecture Error Structure

This is a vendor specific structure. Please refer to your hardware vendor documentation for the format of this structure.

N.2.4.4.2 ARM Processor Context Information

As described above, the processor error section contains a collection of structures called Processor Context Information. These provide processor context state specific to the ARM processor architecture. This section details the layout of the Processor Error Context Information Header Structure (Table N-21) and the detailed processor context type information structures (Table N-21 - Table N-30).

Care must be taken when reporting context information structures. The amount of context reported depends on the agent that is going to observe the data. The following are recommended guidelines:

1. If the error happens whilst the processor is in the secure world, EL3, Secure EL1 or secure EL0, context information can contain sensitive data, and should not be exposed to unauthorized parties.

2. If the error information is being provided to a software agent running at EL2, then the context information should only include any registers visible in EL2, e.g. GPR, EL1 and EL2 registers.
3. If the error information is being provided to a software agent running at EL1, then the context information should only include any registers visible in EL1, e.g. GPR, EL1 and registers.

For context information on processor running in AArch64 mode, even though some registers are defined as 4 bytes in length, following tables provide 8 bytes space to account for possible future expansion.

**Table N-21 ARM Processor Error Context Information Header Structure**

<table>
<thead>
<tr>
<th>Name</th>
<th>Byte Offset</th>
<th>Byte Length</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Version</td>
<td>0</td>
<td>2</td>
<td>(revision of this table)</td>
</tr>
<tr>
<td>Register Context Type</td>
<td>2</td>
<td>2</td>
<td>Value indicating the type of processor context state being reported:</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>0 – AArch32 GPRs (General Purpose Registers).</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>1 -- AArch32 EL1 context registers</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>2 -- AArch32 EL2 context registers</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>3 -- AArch32 secure context registers</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>4 – AArch64 GPRs</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>5 -- AArch64 EL1 context registers</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>6 – AArch64 EL2 context registers</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>7 -- AArch64 EL3 context registers</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>8 – Misc. System Register Structure</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>All other values are reserved.</td>
</tr>
<tr>
<td>Register Array Size</td>
<td>4</td>
<td>4</td>
<td>Represents the total size of the array for the Data Type being reported in bytes.</td>
</tr>
<tr>
<td>Register Array</td>
<td>8</td>
<td>N</td>
<td>This field will provide the contents of the actual registers or raw data. The contents of the array depends on the Type, with the structures described in Tables 266 – 274.</td>
</tr>
</tbody>
</table>

**Table N-22 ARMv8 AArch32 GPRs (Type 0)**

<table>
<thead>
<tr>
<th>Byte Offset</th>
<th>Byte Length</th>
<th>Field</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>4</td>
<td>R0</td>
</tr>
<tr>
<td>4</td>
<td>4</td>
<td>R1</td>
</tr>
<tr>
<td>8</td>
<td>4</td>
<td>R2</td>
</tr>
<tr>
<td>12</td>
<td>4</td>
<td>R3</td>
</tr>
<tr>
<td>16</td>
<td>4</td>
<td>R4</td>
</tr>
<tr>
<td>20</td>
<td>4</td>
<td>R5</td>
</tr>
<tr>
<td>24</td>
<td>4</td>
<td>R6</td>
</tr>
<tr>
<td>28</td>
<td>4</td>
<td>R7</td>
</tr>
<tr>
<td>32</td>
<td>4</td>
<td>R8</td>
</tr>
<tr>
<td>36</td>
<td>4</td>
<td>R9</td>
</tr>
<tr>
<td>40</td>
<td>4</td>
<td>R10</td>
</tr>
<tr>
<td>44</td>
<td>4</td>
<td>R11</td>
</tr>
<tr>
<td>48</td>
<td>4</td>
<td>R12</td>
</tr>
</tbody>
</table>
### Table N-23 ARM AArch32 EL1 Context System Registers (Type 1)

<table>
<thead>
<tr>
<th>Byte Offset</th>
<th>Byte Length</th>
<th>Field</th>
</tr>
</thead>
<tbody>
<tr>
<td>52</td>
<td>4</td>
<td>R13 (SP)</td>
</tr>
<tr>
<td>56</td>
<td>4</td>
<td>R14 (LR)</td>
</tr>
<tr>
<td>60</td>
<td>4</td>
<td>R15 (PC)</td>
</tr>
</tbody>
</table>

### Table N-24 ARM AArch32 EL2 Context System Registers (Type 2)

<table>
<thead>
<tr>
<th>Byte Offset</th>
<th>Byte Length</th>
<th>Field</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>4</td>
<td>ELR_hyp</td>
</tr>
<tr>
<td>4</td>
<td>4</td>
<td>HAMAIR0</td>
</tr>
<tr>
<td>8</td>
<td>4</td>
<td>HAMAIR1</td>
</tr>
</tbody>
</table>
### Table N-25 ARM AArch32 secure Context System Registers (Type 3)

<table>
<thead>
<tr>
<th>Byte Offset</th>
<th>Byte Length</th>
<th>Field</th>
</tr>
</thead>
<tbody>
<tr>
<td>12</td>
<td>4</td>
<td>HCR</td>
</tr>
<tr>
<td>16</td>
<td>4</td>
<td>HCR2</td>
</tr>
<tr>
<td>20</td>
<td>4</td>
<td>HDFAR</td>
</tr>
<tr>
<td>24</td>
<td>4</td>
<td>HIFAR</td>
</tr>
<tr>
<td>28</td>
<td>4</td>
<td>HPFAR</td>
</tr>
<tr>
<td>32</td>
<td>4</td>
<td>HSR</td>
</tr>
<tr>
<td>36</td>
<td>4</td>
<td>HTCR</td>
</tr>
<tr>
<td>40</td>
<td>4</td>
<td>HTPIDR</td>
</tr>
<tr>
<td>44</td>
<td>4</td>
<td>HTTBR</td>
</tr>
<tr>
<td>48</td>
<td>4</td>
<td>SPSR_hyp</td>
</tr>
<tr>
<td>52</td>
<td>4</td>
<td>VTCR</td>
</tr>
<tr>
<td>56</td>
<td>4</td>
<td>VTTBR</td>
</tr>
<tr>
<td>60</td>
<td>4</td>
<td>DACR32_EL2</td>
</tr>
</tbody>
</table>

### Table N-26 ARMv8 AArch64 GPRs (Type 4)

<table>
<thead>
<tr>
<th>Byte Offset</th>
<th>Byte Length</th>
<th>Field</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>8</td>
<td>X0</td>
</tr>
<tr>
<td>8</td>
<td>8</td>
<td>X1</td>
</tr>
<tr>
<td>16</td>
<td>8</td>
<td>X2</td>
</tr>
<tr>
<td>24</td>
<td>8</td>
<td>X3</td>
</tr>
<tr>
<td>32</td>
<td>8</td>
<td>X4</td>
</tr>
<tr>
<td>40</td>
<td>8</td>
<td>X5</td>
</tr>
<tr>
<td>48</td>
<td>8</td>
<td>X6</td>
</tr>
<tr>
<td>56</td>
<td>8</td>
<td>X7</td>
</tr>
<tr>
<td>64</td>
<td>8</td>
<td>X8</td>
</tr>
<tr>
<td>72</td>
<td>8</td>
<td>X9</td>
</tr>
<tr>
<td>80</td>
<td>8</td>
<td>X10</td>
</tr>
<tr>
<td>88</td>
<td>8</td>
<td>X11</td>
</tr>
<tr>
<td>96</td>
<td>8</td>
<td>X12</td>
</tr>
<tr>
<td>104</td>
<td>8</td>
<td>X13</td>
</tr>
</tbody>
</table>
### Table N-27 ARM AArch64 EL1 Context System Registers (Type 5)

<table>
<thead>
<tr>
<th>Byte Offset</th>
<th>Byte Length</th>
<th>Field</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>8</td>
<td>ELR_EL1</td>
</tr>
<tr>
<td>8</td>
<td>8</td>
<td>ESR_EL1</td>
</tr>
<tr>
<td>16</td>
<td>8</td>
<td>FAR_EL1</td>
</tr>
<tr>
<td>24</td>
<td>8</td>
<td>ISR_EL1</td>
</tr>
<tr>
<td>32</td>
<td>8</td>
<td>MAIR_EL1</td>
</tr>
<tr>
<td>40</td>
<td>8</td>
<td>MIDR_EL1</td>
</tr>
<tr>
<td>48</td>
<td>8</td>
<td>MPIDR_EL1</td>
</tr>
<tr>
<td>56</td>
<td>8</td>
<td>SCTLR_EL1</td>
</tr>
<tr>
<td>64</td>
<td>8</td>
<td>SP_EL0</td>
</tr>
<tr>
<td>72</td>
<td>8</td>
<td>SP_EL1</td>
</tr>
<tr>
<td>80</td>
<td>8</td>
<td>SPSR_EL1</td>
</tr>
<tr>
<td>88</td>
<td>8</td>
<td>TCR_EL1</td>
</tr>
<tr>
<td>96</td>
<td>8</td>
<td>TPIDR_EL0</td>
</tr>
<tr>
<td>104</td>
<td>8</td>
<td>TPIDR_EL1</td>
</tr>
<tr>
<td>112</td>
<td>8</td>
<td>TPIDRRO_EL0</td>
</tr>
<tr>
<td>120</td>
<td>8</td>
<td>TTBRO_EL1</td>
</tr>
<tr>
<td>Byte Offset</td>
<td>Byte Length</td>
<td>Field</td>
</tr>
<tr>
<td>-------------</td>
<td>-------------</td>
<td>----------</td>
</tr>
<tr>
<td>128</td>
<td>8</td>
<td>TTBR1_EL1</td>
</tr>
</tbody>
</table>
The following structure (Table 275) describes additional AArch64/AArch32 miscellaneous system registers captured from the perspective of the processor that took the hardware error exception. Each register array entry will be per the following table. The number of register entries present in the register array is based on the register array size (i.e. N/10).

### Table N-28 ARM AArch64 EL2 Context System Registers (Type 6)

<table>
<thead>
<tr>
<th>Byte Offset</th>
<th>Byte Length</th>
<th>Field</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>8</td>
<td>ELR_EL2</td>
</tr>
<tr>
<td>8</td>
<td>8</td>
<td>ESR_EL2</td>
</tr>
<tr>
<td>16</td>
<td>8</td>
<td>FAR_EL2</td>
</tr>
<tr>
<td>24</td>
<td>8</td>
<td>HACR_EL2</td>
</tr>
<tr>
<td>32</td>
<td>8</td>
<td>HCR_EL2</td>
</tr>
<tr>
<td>40</td>
<td>8</td>
<td>HPFAR_EL2</td>
</tr>
<tr>
<td>48</td>
<td>8</td>
<td>MAIR_EL2</td>
</tr>
<tr>
<td>56</td>
<td>8</td>
<td>SCTLR_EL2</td>
</tr>
<tr>
<td>64</td>
<td>8</td>
<td>SP_EL2</td>
</tr>
<tr>
<td>72</td>
<td>8</td>
<td>SPSR_EL2</td>
</tr>
<tr>
<td>80</td>
<td>8</td>
<td>TCR_EL2</td>
</tr>
<tr>
<td>88</td>
<td>8</td>
<td>TPIDR_EL2</td>
</tr>
<tr>
<td>96</td>
<td>8</td>
<td>TTBR0_EL2</td>
</tr>
<tr>
<td>104</td>
<td>8</td>
<td>VTCR_EL2</td>
</tr>
<tr>
<td>112</td>
<td>8</td>
<td>VTTBR_EL2</td>
</tr>
</tbody>
</table>

### Table N-29 ARM AArch64 EL3 Context System Registers (Type 7)

<table>
<thead>
<tr>
<th>Byte Offset</th>
<th>Byte Length</th>
<th>Field</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>8</td>
<td>ELR_EL3</td>
</tr>
<tr>
<td>8</td>
<td>8</td>
<td>ESR_EL3</td>
</tr>
<tr>
<td>16</td>
<td>8</td>
<td>FAR_EL3</td>
</tr>
<tr>
<td>24</td>
<td>8</td>
<td>MAIR_EL3</td>
</tr>
<tr>
<td>32</td>
<td>8</td>
<td>SCTLR_EL3</td>
</tr>
<tr>
<td>40</td>
<td>8</td>
<td>SP_EL3</td>
</tr>
<tr>
<td>48</td>
<td>8</td>
<td>SPSR_EL3</td>
</tr>
<tr>
<td>56</td>
<td>8</td>
<td>TCR_EL3</td>
</tr>
<tr>
<td>64</td>
<td>8</td>
<td>TPIDR_EL3</td>
</tr>
<tr>
<td>72</td>
<td>8</td>
<td>TTBR0_EL3</td>
</tr>
</tbody>
</table>
Table N-30 ARM Misc. Context System Register (Type 8) – Single Register Entry

<table>
<thead>
<tr>
<th>Name</th>
<th>Byte Offset</th>
<th>Byte Length</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>MRS encoding</td>
<td>0</td>
<td>2</td>
<td>This field defines MRS instruction encoding.</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>Bit 0:2 -- Op2</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>Bit 3:6 – CRm</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>Bit 7:10 – CRn</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>Bit 11:13 – Op1</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>Bit 14 – O0</td>
</tr>
<tr>
<td>Value</td>
<td>2</td>
<td>8</td>
<td>Value read from system register</td>
</tr>
</tbody>
</table>

N.2.5 Memory Error Section
Type: {0xA5BC1114, 0x6F64, 0x4EDE, {0xB8, 0x63, 0x3E, 0x83, 0xED, 0x7C, 0x83, 0xB1}}
## Table N-31 Memory Error Record

<table>
<thead>
<tr>
<th>Mnemonic</th>
<th>Byte Offset</th>
<th>Byte Length</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Validation Bits</td>
<td>0</td>
<td>8</td>
<td>Indicates which fields in the memory error record are valid. Bit 0 – Error Status Valid Bit 1 – Physical Address Valid Bit 2 – Physical Address Mask Valid Bit 3 – Node Valid Bit 4 – Card Valid Bit 5 – Module Valid Bit 6 – Bank Valid (When Bank is addressed via group/address, refer to Bit 19 and 20) Bit 7 – Device Valid Bit 8 – Row Valid 1 - the Row field at Offset 42 contains row number (15:0) and row number (17:16) are 00b 0 - the Row field at Offset 42 is not used, or is defined by Bit 18 (Extended Row Bit 16 and 17 Valid). Bit 9 – Column Valid Bit 10 – Bit Position Valid Bit 11 – Platform Requestor Id Valid Bit 12 – Platform Responder Id Valid Bit 13 – Memory Platform Target Valid Bit 14 – Memory Error Type Valid Bit 15 – Rank Number Valid Bit 16 – Card Handle Valid Bit 17 – Module Handle Valid Bit 18 - Extended Row Bit 16 and 17 Valid (refer to Byte Offset 42 and 73 below) 1 - the Row field at Offset 42 contains row number (15:0) and the Extended field at Offset 73 contains row number (17:16) 0 - the Extended field at Offset 73 and the Row field at Offset 42 are not used, or the Row field at Offset 42 is defined by Bit 8 (Row Valid). When this bit is set to 1, Bit 8 (Row Valid) must be set to 0. Bit 19 - Bank Group Valid Bit 20 - Bank Address Valid Bit 21 - Chip Identification Valid Bit 22-63 Reserved</td>
</tr>
<tr>
<td>Error Status</td>
<td>8</td>
<td>8</td>
<td>Memory error status information. See Section N.2.1.2 for error status details.</td>
</tr>
<tr>
<td>Physical Address</td>
<td>16</td>
<td>8</td>
<td>The physical address at which the memory error occurred.</td>
</tr>
<tr>
<td>Physical Address Mask</td>
<td>24</td>
<td>8</td>
<td>Defines the valid address bits in the Physical Address field. The mask specifies the granularity of the physical address which is dependent on the hw/implementation factors such as interleaving.</td>
</tr>
<tr>
<td>Node</td>
<td>32</td>
<td>2</td>
<td>In a multi-node system, this value identifies the node containing the memory in error.</td>
</tr>
<tr>
<td>Card</td>
<td>34</td>
<td>2</td>
<td>The card number of the memory error location.</td>
</tr>
<tr>
<td>Mnemonic</td>
<td>Byte Offset</td>
<td>Byte Length</td>
<td>Description</td>
</tr>
<tr>
<td>--------------</td>
<td>-------------</td>
<td>-------------</td>
<td>------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------</td>
</tr>
<tr>
<td>Module</td>
<td>36</td>
<td>2</td>
<td>The module or rank number of the memory error location. (NODE, CARD, and MODULE should provide the information necessary to identify the failing FRU).</td>
</tr>
<tr>
<td>Bank</td>
<td>38</td>
<td>2</td>
<td>The bank number of the memory associated with the error. When Bank is addressed via group/address Bit 7:0 - Bank Address Bit 15:8 - Bank Group</td>
</tr>
<tr>
<td>Device</td>
<td>40</td>
<td>2</td>
<td>The device number of the memory associated with the error.</td>
</tr>
<tr>
<td>Row</td>
<td>42</td>
<td>2</td>
<td>First 16 bits (15:0) of the row number of the memory error location. This field is valid if either &quot;Row Valid&quot; or &quot;Extended Row Bit 16 and 17&quot; Validation Bits at Offset 0 is set to 1.</td>
</tr>
<tr>
<td>Column</td>
<td>44</td>
<td>2</td>
<td>The column number of the memory error location.</td>
</tr>
<tr>
<td>Bit Position</td>
<td>46</td>
<td>2</td>
<td>The bit position at which the memory error occurred.</td>
</tr>
<tr>
<td>Requestor ID</td>
<td>48</td>
<td>8</td>
<td>Hardware address of the device that initiated the transaction that took the error.</td>
</tr>
<tr>
<td>Responder ID</td>
<td>56</td>
<td>8</td>
<td>Hardware address of the device that responded to the transaction.</td>
</tr>
<tr>
<td>Target ID</td>
<td>64</td>
<td>8</td>
<td>Hardware address of the intended target of the transaction.</td>
</tr>
<tr>
<td>Memory Error Type</td>
<td>72</td>
<td>1</td>
<td>Identifies the type of error that occurred: 0 – Unknown 1 – No error 2 – Single-bit ECC 3 – Multi-bit ECC 4 – Single-symbol ChipKill ECC 5 – Multi-symbol ChipKill ECC 6 – Master abort 7 – Target abort 8 – Parity Error 9 – Watchdog timeout 10 – Invalid address 11 – Mirror Broken 12 – Memory Sparing 13 - Scrub corrected error 14 - Scrub uncorrected error 15 - Physical Memory Map-out event All other values reserved.</td>
</tr>
<tr>
<td>Extended</td>
<td>73</td>
<td>1</td>
<td>Bit 0 - Bit 16 of the row number of the memory error location. This field is valid if &quot;Extended Row Bit 16 and 17&quot; Validation Bits at Offset 0 is set to 1.</td>
</tr>
<tr>
<td>Rank Number</td>
<td>74</td>
<td>2</td>
<td>The Rank number of the memory error location.</td>
</tr>
<tr>
<td>Card Handle</td>
<td>76</td>
<td>2</td>
<td>If bit 16 in Validation Bits is 1, this field contains the SMBIOS handle for the Type 16 Memory Array Structure that represents the memory card.</td>
</tr>
</tbody>
</table>
N.2.6 Memory Error Section 2

Type: { 0x61EC04FC, 0x48E6, 0xD813, { 0x25, 0xC9, 0x8D, 0xAA, 0x44, 0x75, 0x0B, 0x12 } };  
Table N-32 Memory Error Record 2

<table>
<thead>
<tr>
<th>Mnemonic</th>
<th>Byte Offset</th>
<th>Byte Length</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Module Handle</td>
<td>78</td>
<td>2</td>
<td>If bit 17 in Validation Bits is 1, this field contains the SMBIOS handle for the Type 17 Memory Device Structure that represents the Memory Module.</td>
</tr>
<tr>
<td>Validation Bits</td>
<td>0</td>
<td>8</td>
<td>Indicates which fields in the memory error record are valid.</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>Bit 0 – Error Status Valid</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>Bit 1 – Physical Address Valid</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>Bit 2 – Physical Address Mask Valid</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>Bit 3 – Node Valid</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>Bit 4 – Card Valid</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>Bit 5 – Module Valid</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>Bit 6 – Bank Valid (When Bank is addressed via group/address, refer to Bit 20 and 21)</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>Bit 7 – Device Valid</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>Bit 8 – Row Valid</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>Bit 9 – Column Valid</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>Bit 10 - Rank Valid</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>Bit 11 – Bit Position Valid</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>Bit 12 – Chip Identification Valid</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>Bit 13 – Memory Error Type Valid</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>Bit 14 - Status Valid</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>Bit 15 – Requestor ID Valid</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>Bit 16 – Responder ID Valid</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>Bit 17 – Target ID Valid</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>Bit 18 - Card Handle Valid</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>Bit 19 - Module Handle Valid</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>Bit 20 – Bank Group Valid</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>Bit 21 – Bank Address Valid</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>Bit 22-63 Reserved</td>
</tr>
<tr>
<td>Error Status</td>
<td>8</td>
<td>8</td>
<td>Memory error status information. See Section N.2.1.2 for error status details.</td>
</tr>
<tr>
<td>Physical Address</td>
<td>16</td>
<td>8</td>
<td>The physical address at which the memory error occurred.</td>
</tr>
<tr>
<td>Physical Address Mask</td>
<td>24</td>
<td>8</td>
<td>Defines the valid address bits in the Physical Address field. The mask specifies the granularity of the physical address which is dependent on the hardware implementation factors such as interleaving.</td>
</tr>
<tr>
<td>Node</td>
<td>32</td>
<td>2</td>
<td>In a multi-node system, this value identifies the node containing the memory in error.</td>
</tr>
<tr>
<td>Card</td>
<td>34</td>
<td>2</td>
<td>The card number of the memory error location.</td>
</tr>
<tr>
<td>Module</td>
<td>36</td>
<td>2</td>
<td>The module number of the memory error location. (NODE, CARD, and MODULE should provide the information necessary to identify the failing FRU).</td>
</tr>
</tbody>
</table>
N.2.7 PCI Express Error Section

Type: {0xD995E954, 0xBB1, 0x430F, {0xAD, 0x91, 0xB4, 0x4D, 0xCB, 0x3C, 0x6F, 0x35}}
<table>
<thead>
<tr>
<th>Mnemonic</th>
<th>Byte Offset</th>
<th>Byte Length</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Validation Bits</td>
<td>0</td>
<td>8</td>
<td>Indicates which of the following fields is valid:</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>Bit 0 – Port Type Valid</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>Bit 1 – Version Valid</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>Bit 2 – Command Status Valid</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>Bit 3 – Device ID Valid</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>Bit 4 – Device Serial Number Valid</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>Bit 5 – Bridge Control Status Valid</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>Bit 6 – Capability Structure Status Valid</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>Bit 7 – AER Info Valid</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>Bit 8-63 – Reserved</td>
</tr>
<tr>
<td>Port Type</td>
<td>8</td>
<td>4</td>
<td>PCIe Device/Port Type as defined in the PCI Express capabilities register:</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>0: PCI Express End Point</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>1: Legacy PCI End Point Device</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>4: Root Port</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>5: Upstream Switch Port</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>6: Downstream Switch Port</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>7: PCI Express to PCI/PCI-X Bridge</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>8: PCI/PCI-X to PCI Express Bridge</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>9: Root Complex Integrated Endpoint Device</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>10: Root Complex Event Collector</td>
</tr>
<tr>
<td>Version</td>
<td>12</td>
<td>4</td>
<td>PCIe Spec. version supported by the platform:</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>Byte 0-1: PCIe Spec. Version Number</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>• Byte0: Minor Version in BCD</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>• Byte1: Major Version in BCD</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>Byte2-3: Reserved</td>
</tr>
<tr>
<td>Command Status</td>
<td>16</td>
<td>4</td>
<td>Byte0-1: PCI Command Register</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>Byte2-3: PCI Status Register</td>
</tr>
<tr>
<td>Reserved</td>
<td>20</td>
<td>4</td>
<td>Must be zero</td>
</tr>
<tr>
<td>Device ID</td>
<td>24</td>
<td>16</td>
<td>PCIe Root Port PCI/bridge PCI compatible device number and bus number information to uniquely identify the root port or bridge. Default values for both the bus numbers is zero.</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>Byte 0-1: Vendor ID</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>Byte 2-3: Device ID</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>Byte 4-6: Class Code</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>Byte 7: Function Number</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>Byte 8: Device Number</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>Byte 9-10: Segment Number</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>Byte 11: Root Port/Bridge Primary Bus Number or device bus number</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>Byte 12: Root Port/Bridge Secondary Bus Number</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>Byte 13-14: Bit0:2: Reserved Bit3:15 Slot Number</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>Byte 15 Reserved</td>
</tr>
<tr>
<td>Device Serial Number</td>
<td>40</td>
<td>8</td>
<td>Byte 0-3: PCIe Device Serial Number Lower DW</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>Byte 4-7: PCIe Device Serial Number Upper DW</td>
</tr>
<tr>
<td>Bridge Control Status</td>
<td>48</td>
<td>4</td>
<td>This field is valid for bridges only.</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>Byte 0-1: Bridge Secondary Status Register</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>Byte 2-3: Bridge Control Register</td>
</tr>
</tbody>
</table>
N.2.8 PCI/PCI-X Bus Error Section

Type: {0xC5753963, 0x3B84, 0x4095, {0xBF, 0x78, 0xED, 0xDA, 0xD3, 0xF9, 0xC9, 0xDD}}

Table N-34 PCI/PCI-X Bus Error Section

<table>
<thead>
<tr>
<th>Mnemonic</th>
<th>Byte Offset</th>
<th>Byte Length</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Capability Structure</td>
<td>52</td>
<td>60</td>
<td>PCIe Capability Structure.</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>• The 60-byte structure is used to report device capabilities. This</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>structure is used to report the 36-byte PCIe 1.1 Capability</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>Structure (See Figure 7-9 of the PCI Express Base Specification,</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>Rev 1.1) with the last 24 bytes padded.</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>• This structure is also used to report the 60-byte PCIe 2.0</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>Capability Structure (See Figure 7-9 of the PCI Express 2.0 Base</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>Specification.)</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>• The fields in the structure vary with different device types.</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>• The &quot;Next CAP pointer&quot; field should be considered invalid and</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>any reserved fields of the structure are reserved for future use.</td>
</tr>
<tr>
<td>Note that PCIe</td>
<td></td>
<td></td>
<td>devices without AER (PCIe_AER_INFO_STRUCT_VALID_BIT=0) may report</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>status using this structure.</td>
</tr>
<tr>
<td>AER Info</td>
<td>112</td>
<td>96</td>
<td>PCIe Advanced Error Reporting Extended Capability Structure.</td>
</tr>
</tbody>
</table>

Validation Bits

<table>
<thead>
<tr>
<th>Mnemonic</th>
<th>Byte Offset</th>
<th>Byte Length</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Error Status</td>
<td>8</td>
<td>8</td>
<td>PCI Bus Error Status. See Section N.2.1.2 for details.</td>
</tr>
</tbody>
</table>

Error Type

<table>
<thead>
<tr>
<th>Mnemonic</th>
<th>Byte Offset</th>
<th>Byte Length</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Byte 0:</td>
<td>0</td>
<td>8</td>
<td>Indicates which of the following fields is valid:</td>
</tr>
<tr>
<td>Bit 0 –</td>
<td></td>
<td></td>
<td>Error Status Valid</td>
</tr>
<tr>
<td>Bit 1 –</td>
<td></td>
<td></td>
<td>Error Type Valid</td>
</tr>
<tr>
<td>Bit 2 –</td>
<td></td>
<td></td>
<td>Bus Id Valid</td>
</tr>
<tr>
<td>Bit 3 –</td>
<td></td>
<td></td>
<td>Bus Address Valid</td>
</tr>
<tr>
<td>Bit 4 –</td>
<td></td>
<td></td>
<td>Bus Data Valid</td>
</tr>
<tr>
<td>Bit 5 –</td>
<td></td>
<td></td>
<td>Command Valid</td>
</tr>
<tr>
<td>Bit 6 –</td>
<td></td>
<td></td>
<td>Requestor Id Valid</td>
</tr>
<tr>
<td>Bit 7 –</td>
<td></td>
<td></td>
<td>Completer Id Valid</td>
</tr>
<tr>
<td>Bit 8 –</td>
<td></td>
<td></td>
<td>Target Id Valid</td>
</tr>
<tr>
<td>Bit 9-63</td>
<td></td>
<td></td>
<td>Reserved</td>
</tr>
</tbody>
</table>

Byte 1:  

<table>
<thead>
<tr>
<th>Error Type</th>
<th>Byte Offset</th>
<th>Byte Length</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>16</td>
<td>2</td>
<td>PCI Bus error Type</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>Byte 0:</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>0 – Unknown or OEM system specific error</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>1 – Data Parity Error</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>2 – System Error</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>3 – Master Abort</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>4 – Bus Timeout or No Device Present (No DEVSEL#)</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>5 – Master Data Parity Error</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>6 – Address Parity Error</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>7 – Command Parity Error</td>
</tr>
<tr>
<td>Others –</td>
<td></td>
<td></td>
<td>Reserved</td>
</tr>
<tr>
<td>Reserved</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

| Byte 1:  |             |             | Reserved                                                                     |
### N.2.9 PCI/PCI-X Component Error Section

Type: {0xEB5E4685, 0xCA66, 0x4769, {0xB6, 0xA2, 0x26, 0x06, 0x8B, 0x00, 0x13, 0x26}}

#### Table N-35 PCI/PCI-X Component Error Section

<table>
<thead>
<tr>
<th>Mnemonic</th>
<th>Byte Offset</th>
<th>Byte Length</th>
<th>Description</th>
</tr>
</thead>
</table>
| Bus Id            | 18          | 2           | Bits 0:7 – Bus Number  
Bits 8:15 – Segment Number                                                                                                                                                                             |
| Reserved          | 20          | 4           |                                                                                                                                                                                                             |
| Bus Address       | 24          | 8           | Memory or I/O address on the bus at the time of the error.                                                                                                                                                   |
| Bus Data          | 32          | 8           | Data on the PCI bus at the time of the error.                                                                                                                                                              |
| Bus Command       | 40          | 8           | Bus command or operation at the time of the error.  
Byte 7: Bits 7-1: Reserved (should be zero)  
Byte 7: Bit 0: If 0, then the command is a PCI command. If 1, the command is a PCI-X command.                                                                                                                                 |
| Bus Requestor Id  | 48          | 8           | PCI Bus Requestor Id.                                                                                                                                                                                       |
| Bus Completer Id  | 56          | 8           | PCI Bus Responder Id.                                                                                                                                                                                        |
| Target Id         | 64          | 8           | PCI Bus intended target identifier.                                                                                                                                                                          |

<table>
<thead>
<tr>
<th>Mnemonic</th>
<th>Byte Offset</th>
<th>Byte Length</th>
<th>Description</th>
</tr>
</thead>
</table>
| Validation Bits   | 0           | 8           | Indicate which fields are valid:  
Bit 0 – Error Status Valid  
Bit 1 – Id Info Valid  
Bit 2 – Memory Number Valid  
Bit 3 – IO Number Valid  
Bit 4 – Register Data Pair Valid  
Bit 5-63 Reserved                                                                                                                                 |
| Error Status      | 8           | 8           | PCI Component Error Status. See Section N.2.1.2 for details.                                                                                                                                                 |
| Id Info           | 16          | 16          | Identification Information:  
Bytes 0-1: Vendor Id  
Bytes 1-2: Device Id  
Bytes 4-6: Class Code  
Byte 7: Function Number  
Byte 8: Device Number  
Byte 9: Bus Number  
Byte 10: Segment Number  
Bytes 11-15: Reserved                                                                                                                                 |
| Memory Number     | 32          | 4           | Number of PCI Component Memory Mapped register address/data pair values present in this structure.                                                                                                          |
| IO Number         | 36          | 4           | Number of PCI Component Programmed IO register address/data pair values present in this structure.                                                                                                          |
N.2.10 Firmware Error Record Reference

Type: {0x81212A96, 0x09ED, 0x4996, {0x94, 0x71, 0x8D, 0x72, 0x9C, 0x8E, 0x69, 0xED}}

Table N-36 Firmware Error Record Reference

<table>
<thead>
<tr>
<th>Mnemonic</th>
<th>Byte Offset</th>
<th>Byte Length</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Register Data Pairs</td>
<td>40</td>
<td>2x8xN</td>
<td>An array of address/data pair values. The address and data information may be from 2 to 8 bytes of actual data represented in the 8 byte array locations.</td>
</tr>
</tbody>
</table>

N.2.11 DMAr Error Sections

The DMAr error sections are divided into two different components as described below:

**DMAr Generic Error Section:**

This section holds information about DMAr errors in a generic form and will be common across all DMAr unit architectures.

**Architecture specific DMAr Error Section:**

This section consists of DMA remapping errors specific to the architecture. In addition, certain state information of the DMAr unit is captured at the time of error. This section is unique for each DMAr architecture (VT-d, IOMMU).
### N.2.11.1 DMAr Generic Error Section

Type: `{0x5B51FEF7, 0xC79D, 0x4434, {0x8F, 0x1B, 0xAA, 0x62, 0xDE, 0x3E, 0x2C, 0x64}}`

#### Table N-37 DMAr Generic Errors

<table>
<thead>
<tr>
<th>Mnemonic</th>
<th>Byte Offset</th>
<th>Byte Length</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Requester-ID</td>
<td>0</td>
<td>2</td>
<td>Device ID associated with a fault condition</td>
</tr>
<tr>
<td>Segment Number</td>
<td>2</td>
<td>2</td>
<td>PCI segment associated with a device</td>
</tr>
<tr>
<td>Fault Reason</td>
<td>4</td>
<td>1</td>
<td>1h: Domain mapping table entry is not present</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>2h: Invalid domain mapping table entry</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>3h: DMAr unit’s attempt to access the domain mapping table resulted in an error</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>4h: Reserved bit set to non-zero value in the domain mapping table</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>5h: DMA request to access an address beyond the device address width</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>6h: Invalid read or write access</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>7h: Invalid device request</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>8h: DMAr unit’s attempt to access the address translation table resulted in an error</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>9h: Reserved bit set to non-zero value in the address translation table</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>Ah: Illegal command error</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>Bh: DMAr unit’s attempt to access the command buffer resulted in an error</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>Other values are reserved</td>
</tr>
<tr>
<td>Access Type</td>
<td>5</td>
<td>1</td>
<td>0h: DMA Write</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>1h: DMA Read</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>Other values are reserved</td>
</tr>
<tr>
<td>Address Type</td>
<td>6</td>
<td>1</td>
<td>0h: Untranslated request</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>1h: Translation request</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>Other values are reserved</td>
</tr>
<tr>
<td>Architecture Type</td>
<td>7</td>
<td>1</td>
<td>1h: VT-d architecture</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>2h: IOMMU architecture</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>Other values are reserved</td>
</tr>
<tr>
<td>Device Address</td>
<td>8</td>
<td>8</td>
<td>This field contains the 64-bit device virtual address in the faulted DMA request.</td>
</tr>
<tr>
<td>Reserved</td>
<td>16</td>
<td>16</td>
<td>Must be 0</td>
</tr>
</tbody>
</table>
N.2.11.2 Intel® VT for Directed I/O specific DMAr Error Section

Type: \{0x71761D37, 0x32B2, 0x45cd, \{0xA7, 0xD0, 0xB0, 0xFE, 0xDD, 0x93, 0xE8, 0xCF\}\}

All fields in this error section are specific to Intel’s VT-d architecture. This error section has a fixed size.

**Table N-38 Intel® VT for Directed I/O specific DMAr Errors**

<table>
<thead>
<tr>
<th>Mnemonic</th>
<th>Byte Offset</th>
<th>Byte Length</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Version</td>
<td>0</td>
<td>1</td>
<td>Value of version register as defined in VT-d architecture</td>
</tr>
<tr>
<td>Revision</td>
<td>1</td>
<td>1</td>
<td>Value of revision field in VT-d specific DMA remapping reporting structure</td>
</tr>
<tr>
<td>OemId</td>
<td>2</td>
<td>6</td>
<td>Value of OEM ID field in VT-d specific DMA remapping reporting structure</td>
</tr>
<tr>
<td>Capability</td>
<td>8</td>
<td>8</td>
<td>Value of capability register in VT-d architecture</td>
</tr>
<tr>
<td>Extended Capability</td>
<td>16</td>
<td>8</td>
<td>Value of extended capability register in VT-d architecture</td>
</tr>
<tr>
<td>Global Command</td>
<td>24</td>
<td>4</td>
<td>Value of Global Command register in VT-d architecture programmed by the operating system</td>
</tr>
<tr>
<td>Global Status</td>
<td>28</td>
<td>4</td>
<td>Value of Global Status register in VT-d architecture</td>
</tr>
<tr>
<td>Fault Status</td>
<td>32</td>
<td>4</td>
<td>Value of Fault Status register in VT-d architecture</td>
</tr>
<tr>
<td>Reserved</td>
<td>36</td>
<td>12</td>
<td>Must be 0</td>
</tr>
<tr>
<td>Fault record</td>
<td>48</td>
<td>16</td>
<td>Fault record as defined in the VT-d specification</td>
</tr>
<tr>
<td>Root Entry</td>
<td>64</td>
<td>16</td>
<td>Value from the root entry table for the given requester-ID</td>
</tr>
<tr>
<td>Context Entry</td>
<td>80</td>
<td>16</td>
<td>Value from the context entry table for the given requester-ID.</td>
</tr>
<tr>
<td>Level 6 Page Table Entry</td>
<td>96</td>
<td>8</td>
<td>PTE entry for device virtual address in page level 6</td>
</tr>
<tr>
<td>Level 5 Page Table Entry</td>
<td>104</td>
<td>8</td>
<td>PTE entry for device virtual address in page level 5</td>
</tr>
<tr>
<td>Level 4 Page Table Entry</td>
<td>112</td>
<td>8</td>
<td>PTE entry for device virtual address in page level 4</td>
</tr>
<tr>
<td>Level 3 Page Table Entry</td>
<td>120</td>
<td>8</td>
<td>PTE entry for device virtual address in page level 3</td>
</tr>
<tr>
<td>Level 2 Page Table Entry</td>
<td>128</td>
<td>8</td>
<td>PTE entry for device virtual address in page level 2.</td>
</tr>
<tr>
<td>Level 1 Page Table Entry</td>
<td>136</td>
<td>8</td>
<td>PTE entry for device virtual address in page level 1</td>
</tr>
</tbody>
</table>
N.2.11.3 IOMMU specific DMAr Error Section

Type: {0x036F84E1, 0x7F37, 0x428c, {0xA7, 0x9E, 0x57, 0x5F, 0xDF, 0xAA, 0x84, 0xEC}}

All fields in this error record are specific to AMD’s IOMMU specification. This error section has a fixed size.

Table N-39 IOMMU-specific DMAr Errors

<table>
<thead>
<tr>
<th>Mnemonic</th>
<th>Byte Offset</th>
<th>Byte Length</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Revision</td>
<td>0</td>
<td>1</td>
<td>Specifies the IOMMU specification revision</td>
</tr>
<tr>
<td>Reserved</td>
<td>1</td>
<td>7</td>
<td>Must be 0</td>
</tr>
<tr>
<td>Control</td>
<td>8</td>
<td>8</td>
<td>IOMMU control register</td>
</tr>
<tr>
<td>Status</td>
<td>16</td>
<td>8</td>
<td>IOMMU status register</td>
</tr>
<tr>
<td>Reserved</td>
<td>24</td>
<td>8</td>
<td>Must be 0</td>
</tr>
<tr>
<td>Event Log Entry</td>
<td>32</td>
<td>16</td>
<td>IOMMU fault related event log entry as defined in the IOMMU specification</td>
</tr>
<tr>
<td>Reserved</td>
<td>48</td>
<td>16</td>
<td>Must be 0</td>
</tr>
<tr>
<td>Device Table Entry</td>
<td>64</td>
<td>32</td>
<td>Value from the device table for a given Requester ID</td>
</tr>
<tr>
<td>Level 6 Page Table Entry</td>
<td>96</td>
<td>8</td>
<td>PTE entry for device virtual address in page level 6</td>
</tr>
<tr>
<td>Level 5 Page Table Entry</td>
<td>104</td>
<td>8</td>
<td>PTE entry for device virtual address in page level 5</td>
</tr>
<tr>
<td>Level 4 Page Table Entry</td>
<td>112</td>
<td>8</td>
<td>PTE entry for device virtual address in page level 4</td>
</tr>
<tr>
<td>Level 3 Page Table Entry</td>
<td>120</td>
<td>8</td>
<td>PTE entry for device virtual address in page level 3</td>
</tr>
<tr>
<td>Level 2 Page Table Entry</td>
<td>128</td>
<td>8</td>
<td>PTE entry for device virtual address in page level 2</td>
</tr>
<tr>
<td>Level 1 Page Table Entry</td>
<td>136</td>
<td>8</td>
<td>PTE entry for device virtual address in page level 1</td>
</tr>
</tbody>
</table>
N.2.12 CCIX PER Log Error Section

Type: \{0x9133EF6, 0xEBFB, 0x4478, \{0xA6, 0xA6, 0x88, 0xB7, 0x28, 0xCF, 0x75, 0xD7\}\}

Table N-40 CCIX PER Log Error Record

<table>
<thead>
<tr>
<th>Mnemonic</th>
<th>Byte Offset</th>
<th>Byte Length</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Length</td>
<td>0</td>
<td>4</td>
<td>Length in bytes for entire structure.</td>
</tr>
<tr>
<td>Validation Bits</td>
<td>4</td>
<td>8</td>
<td>Indicates which of the following fields is valid:</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>Bit 0 – CCIX Source ID Valid</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>Bit 1 – CCIX Port ID Valid</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>Bit 2 – CCIX PER Log Valid</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>Bit 3-63 – Reserved</td>
</tr>
<tr>
<td>CCIX Source ID</td>
<td>12</td>
<td>1</td>
<td>If the agent type is an HA, SA, or RA: This field indicates the CCIX Agent ID of the component that reported this error. In this case bits 7:6 must be zero, since Agent ID is only 6 bits. Otherwise, this field specifies the CCIX Device ID (i.e. in the case of Port, CCIX Link, or device errors).</td>
</tr>
<tr>
<td>CCIX Port ID</td>
<td>13</td>
<td>1</td>
<td>This field indicates the CCIX Port ID that reported this error. Bits 7:5 must be zero, since CCIX Port ID is only 5 bits.</td>
</tr>
<tr>
<td>Reserved</td>
<td>14</td>
<td>2</td>
<td>Must be zero.</td>
</tr>
<tr>
<td>CCIX PER Log</td>
<td>16</td>
<td>20...n</td>
<td>DWORD (32-bit) entries in CCIX PER Log Structure, as described in Section 7.3.2 of the CCIX Base Specification – Revision 1.0. NOTE: The Per Log Structure contains a header describing the number of DWORDs in the error record.</td>
</tr>
</tbody>
</table>

N.2.13 Compute Express Link (CXL) Protocol Error Section

Type: \{0x80B9EFB4, 0x52B5, 0x4DE3, \{0xA7, 0x77, 0x68, 0x78, 0x4B, 0x77, 0x10, 0x48\}\}

Table N-41 Table 94. CXL Protocol Error Section

<table>
<thead>
<tr>
<th>Mnemonic</th>
<th>Byte Offset</th>
<th>Byte Length</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Validation Bits</td>
<td>0</td>
<td>8</td>
<td>Indicates which of the following fields is valid:</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>Bit 0 – CXL Agent Type field is valid</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>Bit 1 – CXL Agent Address field is valid</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>Bit 2 – Device ID field is valid</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>Bit 3 – Device Serial Number field is valid</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>Bit 4 – Capability Structure field is valid</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>Bit 5 – CXL DVSEC field it valid</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>Bit 6 – CXL Error Log field is valid</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>Bits 7:63 – Reserved</td>
</tr>
<tr>
<td>CXL Agent Type</td>
<td>8</td>
<td>1</td>
<td>0 – This error was detected by a CXL 1.1 device</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>1 – This error was detected by a CXL 1.1 host downstream port</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>2-255 – Reserved</td>
</tr>
<tr>
<td>Reserved</td>
<td>9</td>
<td>7</td>
<td>Must be zero</td>
</tr>
<tr>
<td>Field</td>
<td>Size</td>
<td>alignment</td>
<td>Description</td>
</tr>
<tr>
<td>-----------------------</td>
<td>------</td>
<td>-----------</td>
<td>------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------</td>
</tr>
<tr>
<td>CXL Agent Address</td>
<td>16</td>
<td>8</td>
<td>If CXL agent is a CXL 1.1 device; Byte 0 – Function number, Byte 1 – Device number, Byte 2 – Bus number, Bytes 3-4 – Segment number, Bytes 5-7 - Reserved. If CXL agent is a CXL 1.1 host downstream port, Byte 0-7 – CXL Port RCRB Base address.</td>
</tr>
<tr>
<td>Device ID</td>
<td>24</td>
<td>16</td>
<td>If CXL agent is a CXL 1.1 device, the PCI compatible device number and bus number information to uniquely identify the device. Bytes 0-1: Vendor ID, Bytes 2-3: Device ID, Bytes 4-5: Subsystem Vendor ID, Bytes 6-7: Subsystem Device ID, Bytes 8-9: Class Code, Byte 10-11: Bits 0:2: Reserved, Bits 3:15 Slot Number, Byte 12-15 Reserved.</td>
</tr>
<tr>
<td>Device Serial Number</td>
<td>40</td>
<td>8</td>
<td>If CXL agent is a CXL 1.1 device; Byte 0-3: CXL Device Serial Number Lower DW, Byte 4-7: CXL Device Serial Number Upper DW.</td>
</tr>
<tr>
<td>Capability Structure</td>
<td>48</td>
<td>60</td>
<td>If CXL agent is a CXL 1.1 device, PCIe Capability Structure. The 60-byte structure is used to report device capabilities. This structure is used to report the 36-byte PCIe 1.1 Capability Structure (See Figure 7-9 of the PCI Express Base Specification, Rev 1.1) with the last 24 bytes padded. This structure is also used to report the 60-byte PCIe 2.0 Capability Structure (See Figure 7-9 of the PCI Express 2.0 Base Specification.) The fields in the structure vary with different device types. The &quot;Next CAP pointer&quot; field should be considered invalid and any reserved fields of the structure are reserved for future use. Note that PCIe devices without AER (PCIe_AER_INFO_STRUCT_VALID_BIT=0) may report status using this structure.</td>
</tr>
<tr>
<td>CXL DVSEC Length</td>
<td>108</td>
<td>2</td>
<td>The length in bytes of the CXL DVSEC field.</td>
</tr>
<tr>
<td>CXL Error Log Length</td>
<td>110</td>
<td>2</td>
<td>The length in bytes of the CXL Error Log field.</td>
</tr>
<tr>
<td>Reserved</td>
<td>112</td>
<td>4</td>
<td>Must be zero.</td>
</tr>
<tr>
<td>CXL DVSEC</td>
<td>116</td>
<td>varies</td>
<td>The length of this variable-length structure is defined by the CXL DVSEC Length field. If the CXL agent is a CXL 1.1 device, this field contains a copy of the CXL Device DVSEC, as defined by the &quot;CXL DVSEC for Flex Bus Device&quot; structure in the CXL 1.1 Specification. If the CXL agent is a CXL 1.1 host downstream port, this field contains a copy of the CXL Port DVSEC, as defined by the &quot;CXL DVSEC for Flex Bus Port&quot; structure in CXL 1.1 Specification.</td>
</tr>
<tr>
<td>CXL Error Log</td>
<td>varies</td>
<td>varies</td>
<td>The length of this variable-length structure is defined by the CXL Error Log Length field. For CXL 1.1 devices and host downstream ports, this field contains a copy of the &quot;CXL RAS Capability Structure&quot;, as defined in the CXL 1.1 Specification.</td>
</tr>
</tbody>
</table>
### N.2.14 CXL Component Events Section

Refer to the Events Record Format for CXL components in the CXL Specification, Rev 2.0 or later.

- For the Section Type GUID: Refer to the Event Record Identifier field (Offset 0) of the Events Record Format for each CXL component.
- For the CXL Component Event Log: Refer to the Common Event Record field (Offset 16) of the Events Record Format for each CXL component.

#### Table N-42 CXL Component Event Log Record

<table>
<thead>
<tr>
<th>Mnemonic</th>
<th>Byte Offset</th>
<th>Byte Length</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Length</td>
<td>0</td>
<td>4</td>
<td>Length in bytes for entire structure.</td>
</tr>
<tr>
<td>Validation Bits</td>
<td>4</td>
<td>8</td>
<td>Bit 0 – Device ID Valid</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>Bit 1 – Device Serial Number Valid</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>Bit 2 – CXL Component Event Log Valid</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>Bit 3-63 – Reserved</td>
</tr>
<tr>
<td>Device ID</td>
<td>12</td>
<td>12</td>
<td>PCIe Device Identifiers of CXL Component:</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>Byte 0-1: Vendor ID</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>Byte 2-3: Device ID</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>Byte 4: Function Number</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>Byte 5: Device Number</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>Byte 6: Bus Number</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>Byte 7-8: Segment Number</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>Byte 9-10:</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>Bit 0:2: Reserved</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>Bit 3:15 Slot Number</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>Byte 11 Reserved</td>
</tr>
<tr>
<td>Device Serial Number</td>
<td>24</td>
<td>8</td>
<td>Byte 0-3: PCIe Device Serial Number Lower DW</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>Byte 4-7: PCIe Device Serial Number Upper DW</td>
</tr>
<tr>
<td>CXL Component Event Log</td>
<td>32</td>
<td>+</td>
<td>CXL Component Event Log, starting with the Common Event Record field</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>corresponding to the Component specified by the Section Type GUID. The</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>Length of this field may vary.</td>
</tr>
</tbody>
</table>
Appendix O - UEFI ACPI Data Table

To prevent ACPI namespace collision, a UEFI ACPI table format is defined. This allows creation of ACPI tables without colliding with tables reserved in the namespace.

Table O-1 UEFI Table Structure

<table>
<thead>
<tr>
<th>Field</th>
<th>Byte Length</th>
<th>Byte Offset</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Signature</td>
<td>4</td>
<td>0</td>
<td>‘UEFI’ (0x55, 0x45, 0x46, 0x49). Signature for UEFI drivers that produce ACPI tables.</td>
</tr>
<tr>
<td>Length</td>
<td>4</td>
<td>4</td>
<td>Length, in bytes, of the entire UEFI Table</td>
</tr>
<tr>
<td>Revision</td>
<td>1</td>
<td>8</td>
<td>1</td>
</tr>
<tr>
<td>Checksum</td>
<td>1</td>
<td>9</td>
<td>Entire table must sum to zero.</td>
</tr>
<tr>
<td>OEMID</td>
<td>6</td>
<td>10</td>
<td>OEM ID.</td>
</tr>
<tr>
<td>OEM Table ID</td>
<td>8</td>
<td>16</td>
<td>For the UEFI Table, the table ID is the manufacture model ID.</td>
</tr>
<tr>
<td>OEM Revision</td>
<td>4</td>
<td>24</td>
<td>OEM revision of UEFI table for supplied OEM Table ID.</td>
</tr>
<tr>
<td>Creator ID</td>
<td>4</td>
<td>28</td>
<td>Vendor ID of utility that created the table.</td>
</tr>
<tr>
<td>Creator Revision</td>
<td>4</td>
<td>32</td>
<td>Revision of utility that created the table.</td>
</tr>
<tr>
<td>Identifier</td>
<td>16</td>
<td>36</td>
<td>This value contains a GUID which identifies the remaining table contents.</td>
</tr>
<tr>
<td>DataOffset</td>
<td>2</td>
<td>52</td>
<td>Specifies the byte offset to the remaining data in the UEFI table.</td>
</tr>
<tr>
<td>Data</td>
<td>X</td>
<td>DataOffset</td>
<td>Contains the rest of the UEFI table contents</td>
</tr>
</tbody>
</table>

The first use of this UEFI ACPI table format is the SMM Communication ACPI Table. This table describes a special software SMI that can be used to initiate inter-mode communication in the OS present environment by non-firmware agents with SMM code.

**Note:** The use of the SMM Communication ACPI table is deprecated in UEFI spec. 2.7. This is due to the lack of a use case for inter-mode communication by non-firmware agents with SMM code and support for initiating this form of communication in common OSes.

Table O-2 SMM Communication ACPI Table.

<table>
<thead>
<tr>
<th>Field</th>
<th>Byte Length</th>
<th>Byte Offset</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Signature</td>
<td>4</td>
<td>0</td>
<td>‘UEFI’ (0x55, 0x45, 0x46, 0x49) Signature for UEFI drivers that produce ACPI tables.</td>
</tr>
<tr>
<td>Length</td>
<td>4</td>
<td>4</td>
<td>66+N. Length, in bytes, of the entire Table. N is a length of the optional implementation specific data that can be included in this table.</td>
</tr>
<tr>
<td>Revision</td>
<td>1</td>
<td>8</td>
<td>2</td>
</tr>
<tr>
<td>Checksum</td>
<td>1</td>
<td>9</td>
<td>Entire table must sum to zero.</td>
</tr>
</tbody>
</table>
Invocation method

There are two methods of invocation provided by this specification:

1. Using invocation register

If the invocation register is non-zero, this then this method takes precedence and the SW SMI number field and DataOffset fields must be ignored. The invocation register entry provides the address of a register that must be written to in order to invoke the SMM service. The caller must write the communication buffer address into the register. This will cause an SMM invocation. Upon return from the SMM service call the value in the register provides a return error codes from the SMM invocation. See PI/SMM Vol 4 version xx.yy `EFI_SMM_COMMUNICATION_PROTOCOL.Communicate` function for valid error codes.

<table>
<thead>
<tr>
<th>Field</th>
<th>Byte Length</th>
<th>Byte Offset</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>OEMID</td>
<td>6</td>
<td>10</td>
<td>OEM ID.</td>
</tr>
<tr>
<td>OEM Table ID</td>
<td>8</td>
<td>16</td>
<td>For the UEFI Table, the table ID is the manufacturer model ID.</td>
</tr>
<tr>
<td>OEM Revision</td>
<td>4</td>
<td>24</td>
<td>OEM revision of UEFI table for supplied OEM Table ID.</td>
</tr>
<tr>
<td>Creator ID</td>
<td>4</td>
<td>28</td>
<td>Vendor ID of utility that created the table.</td>
</tr>
<tr>
<td>Creator Revision</td>
<td>4</td>
<td>32</td>
<td>Revision of utility that created the table.</td>
</tr>
<tr>
<td>Identifier</td>
<td>16</td>
<td>36</td>
<td>GUID {0xc68ed8e2, 0x9dc6, 0x4cbd, 0x9d, 0x94, 0xdb, 0x65, \ 0xac, 0xc5, 0xc3, 0x32}</td>
</tr>
<tr>
<td>DataOffset</td>
<td>2</td>
<td>52</td>
<td>Must be 54 for this version of the specification. Specifies the byte offset of the SW SMI Number field, relative to the start of this table. Future expansion may place additional fields between DataOffset and SW SMI Number, so this offset should always be used to calculate the location of SW SMI Number.</td>
</tr>
<tr>
<td>SW SMI Number</td>
<td>4</td>
<td>54</td>
<td>Number to write into software SMI triggering port.</td>
</tr>
<tr>
<td>Buffer Ptr Address</td>
<td>8</td>
<td>58</td>
<td>Address of the communication buffer pointer. The pointer address (this field) and the pointer value (the actual address of the communication buffer) are 64-bit physical addresses. The creator of this table must initialize pointer value with 0. The communication buffer must begin with the <code>EFI_SMM_COMMUNICATE_HEADER</code> defined in the &quot;Related Definitions&quot; section below. The communication buffer must be physically contiguous.</td>
</tr>
<tr>
<td>Invocation register</td>
<td>12</td>
<td>66</td>
<td>Generic Address Structure (GAS) which provides the address of a register that must be written to with the address of a communication buffer to invoke a management mode service. Using this method of invocation is optional, and if not present this span of the table should be populated with zeros. See ACPI6.0 &quot;Generic Address Structure&quot;</td>
</tr>
</tbody>
</table>
The invocation address field uses generic address structure to specify the register address. GAS allows the address space of the register to be Functional Fixed Hardware (FFH). If this address space is used please refer to CPU architecture specific documentation for ascertaining how the write to the register should be performed. For more details on the GAS format please see the ACPI Specification.

Note that for implementations that support concurrent invocation of SMM from multiple processors, the register provided must be a per processor register. In such implementation, the calling execution context must not migrate from one CPU to another between writing to the register, to make the SMM call, and reading the value of the register, to read the error return code.

2. Using the SW SMI number

This method is specific to x86 CPUs.

In order to initiate inter-mode communication OS present agent has to perform the following tasks:

- Prepare communication data buffer that starts with the **EFI_SMM_COMMUNICATE_HEADER**.
- Check the value of the communication buffer pointer (a value at the address specified by the Buffer Ptr Address field). If the pointer's value is zero, update it with the address of the communication buffer. If the pointer's value is non-zero, another inter-mode communication transaction is in progress, and the current communication attempt has to be postponed or canceled.

**Note:** These steps must be performed as an atomic transaction. For example, on IA-32/x64 platforms this can be done using the CMPXCHG CPU instruction.

- Generate software SMI using value from the SMM Communication ACPI Table. The actual means of generating the software SMI is platform-specific.
- Set communication buffer pointer's value to zero.

**Related Definitions**

typedef struct {
    EFI_GUID HeaderGuid;
    UINTN MessageLength;
    UINT8 Data[ANYSIZE_ARRAY];
} EFI_SMM_COMMUNICATE_HEADER;

*HeaderGuid*  
Allows for disambiguation of the message format. Type **EFI_GUID** is defined in `InstallProtocolInterface()`.

*MessageLength*  
Describes the size of **Data** (in bytes) and does not include the size of the header.

*Data*  
Designates an array of bytes that is **MessageLength** in size.
Appendix P - Hardware Error Record Persistence Usage

The OS determines if a platform implements support for Hardware Error Record Persistence by reading the HwErrRecSupport globally defined variable. If the attempt to read this variable returns EFI_NOT_FOUND (14), then the OS will infer that the platform does not implement Hardware Error Record Persistence. If the attempt to read this variable succeeds, then the OS uses the returned value to determine whether the platform supports Hardware Error Record Persistence. A non-zero value indicates that the platform supports Hardware Error Record Persistence.

P.1 Determining space
To determine the amount of space (in bytes) guaranteed by the platform for saving hardware error records, the OS invokes QueryVariableInfo, setting the HR bit in the Attributes bitmask.

P.2 Saving Hardware error records
To save a hardware error record, the OS invokes SetVariable, supplying EFI_HARDWARE_ERROR_VARIABLE as the VendorGuid and setting the HR bit in the Attributes bitmask. The VariableName will be constructed by the OS by concatenating an index to the string “HwErrRec” (i.e., HwErrRec0001). The index portion of the variable name is determined by reading all of the hardware error record variables currently stored on the platform and choosing an appropriate index value based on the names of the existing variables. The platform saves the supplied Data. If insufficient space is present to store the record, the platform will return EFI_OUT_OF_RESOURCES, in which case, the OS may clear an existing record and retry. A retry attempt may continue to fail with status EFI_OUT_OF_RESOURCES if a reboot is required to coalesce resources after deletion. The OS may only save error records after ExitBootServices is called. Firmware may also use the Hardware Error Record Persistence interface to write error records, but it may only do so before ExitBootServices is called. If firmware uses this interface to write an error record, it must use the VariableName format used by the OS as described above and the error records it creates must contain the firmware’s CreatorId. Firmware may overwrite error records whose CreatorId matches the firmware’s CreatorId. Firmware may overwrite error records that have been cleared by other components.

During OS initialization, the OS discovers the names of all persisted error record variables by enumerating the current variable names using GetNextVariableName. Having identified the names of all error record variables, the OS will then read and process all of the error records from the store. After the OS processes an error record, it clears the variable if it was the creator of the variable (determined by checking the CreatorId field of the error record).

P.3 Clearing error record variables
To clear error record variables, the OS invokes SetVariable, supplying EFI_HARDWARE_ERROR_VARIABLE as the VendorGuid and setting the HR bit in the Attributes bitmask. The supplied DataSize, and Data parameters will all be set to zero to indicate that the variable is to be cleared. The supplied VariableName identifies which error record variable is to be cleared. The OS may only clear error records after ExitBootServices has been called. The OS itself may only clear error records which it created (e.g. error records whose CreatorId matches that of the OS). However, a management application running on the OS
may clear error records created by other components. This enables error records created by firmware or other OSes to be cleared by the currently running OS.
Appendix Q - References

Q.1 Related Information
The following publications and sources of information may be useful to you or are referred to by this specification:

- "802.1x Port-based access control” at “Links to UEFI-Related Documents” (http://uefi.org/uefi).
- "802.1X-2004, IEEE Standard for Local and Metropolitan Area Networks Port-Based Network Access Control” at “Links to UEFI-Related Documents” (http://uefi.org/uefi).
- "Advanced Configuration and Power Interface Specification, 4.0” at “Links to UEFI-Related Documents” (http://uefi.org/uefi).
- "Address Resolution Protocol; Refer to Appendix E, “32/64-Bit UNDI Specification” at “Links to UEFI-Related Documents” (http://uefi.org/uefi).
- "CAE Specification [UUID], DCE 1.1:Remote Procedure Call, Document Number C706, Universal Unique Identifier Appendix” at “Links to UEFI-Related Documents” (http://uefi.org/uefi).
- "[BASE64] RFC 1521: MIME (Multipurpose Internet Mail Extensions) Part One: Mechanisms for Specifying and Describing the Format of Internet Message Bodies” at “Links to UEFI-Related Documents” (http://uefi.org/uefi).
- "BLUETOOTH SPECIFICATION, version 4.1” at “Links to UEFI-Related Documents” (http://uefi.org/uefi).
- Bootstrap Protocol; This reference is included for backward compatibility. BC protocol supports DHCP and BOOTP. Refer to Appendix E, “32/64-Bit UNDI Specification, RFC 0951” at “Links to UEFI-Related Documents” (http://uefi.org/uefi).
- Compute Express Link (CXL) Specification: see download link at https://uefi.org/uefi#cxl-specification.
• "EFI Specification Version 1.02" heading at “Links to UEFI-Related Documents” (http://uefi.org/uefi).
• "EFI Specification Version 1.10" heading at “Links to UEFI-Related Documents” (http://uefi.org/uefi).
• "File Verification Using CRC", Mark R. Nelson, Dr. Dobbs, May 1994
• "Information Technology — BIOS Enhanced Disk Drive Services (EDD), working draft T13/1386D, Revision 5a” heading at “Links to UEFI-Related Documents” (http://uefi.org/uefi).
• "Intel 64 and IA-32 Architecture Software Developer’s Manual ” heading at “Links to UEFI-Related Documents” (http://uefi.org/uefi).
• "iSCSI Boot Firmware Table (iBFT) as defined in ACPI 3.0b Specification, Version 1.01,” heading at “Links to UEFI-Related Documents” (http://uefi.org/uefi).
• "ISO Standard 9995, Keyboard layouts for text and office systems” heading at “Links to UEFI-Related Documents” (http://uefi.org/uefi).
• "Itanium® Architecture Software Developer’s Manual, Volume 3: Instruction Set Reference, Rev. 2.2” heading at “Links to UEFI-Related Documents” (http://uefi.org/uefi).
• "Itanium® Software Conventions and Runtime Architecture Guide” heading at “Links to UEFI-Related Documents” (http://uefi.org/uefi).
• "Itanium® System Abstraction Layer Specification” heading at “Links to UEFI-Related Documents” (http://uefi.org/uefi).
• "Internet Engineering Task Force (IETF)” heading at “Links to UEFI-Related Documents” (http://uefi.org/uefi). Refer to Appendix E, “32/64-Bit UNDI Specification,” for more information.
• ITU-T Rec. V.42, Error-Correcting Procedures for DCEs using asynchronous-to-synchronous conversion, October, 1996


• “Microsoft Windows Authenticode Portable Executable Signature Format, Version 1.0” heading at “Links to UEFI-Related Documents” (http://uefi.org/uefi).

• “OSTA Universal Disk Format Specification, Revision 2.00” heading at “Links to UEFI-Related Documents” (http://uefi.org/uefi).


• “Microsoft Windows Authenticode Portable Executable Signature Format, Version 1.0” heading at “Links to UEFI-Related Documents” (http://uefi.org/uefi).


• “Microsoft’s PEAP version 0” at “Links to UEFI-Related Documents” (http://uefi.org/uefi).


• “Processor Architecture Type” at “Links to UEFI-Related Documents” (http://uefi.org/uefi).

• “Protected EAP Protocol (PEAP)” at “Links to UEFI-Related Documents” (http://uefi.org/uefi).

• “Protected EAP Protocol (PEAP) Version 2” at “Links to UEFI-Related Documents” (http://uefi.org/uefi).
• "Request For Comments" heading at “Links to UEFI-Related Documents” (http://uefi.org/uefi).
  Refer to Appendix E, “32/64-Bit UNDI Specification,” for more information.


• "[RFC 1700] ASSIGNED NUMBERS” at “Links to UEFI-Related Documents” (http://uefi.org/uefi).


• "[RFC 2409] The Internet Key Exchange (IKE)” at “Links to UEFI-Related Documents” (http://uefi.org/uefi).


• "[RFC 2459] Internet X.509 Public Key Infrastructure Certificate and CRL Profile” at “Links to UEFI-Related Documents” (http://uefi.org/uefi).


• "[RFC2463] ICMP for IPv6” heading at “Links to UEFI-Related Documents” (http://uefi.org/uefi). Refer to Appendix E, “32/64-Bit UNDI Specification,” for more information


• "[RFC 2818] HTTP Over TLS” at “Links to UEFI-Related Documents” (http://uefi.org/uefi).

• "[RFC 3004] The User Class option for DHCP” at “Links to UEFI-Related Documents” (http://uefi.org/uefi).


• "[RFC 3720] Internet Small Computer Systems Interface (iSCSI)” at “Links to UEFI-Related Documents” (http://uefi.org/uefi).

• “[RFC 4347] Datagram Transport Layer Security” at “Links to UEFI-Related Documents” (http://uefi.org/uefi)
• “Universal Serial Bus Specification, Revision 2.0” heading at “Links to UEFI-Related Documents” (http://uefi.org/uefi).
Q.2 Prerequisite Specifications

In general, this specification requires that functionality defined in a number of other existing specifications be present on a system that implements this specification. This specification requires that those specifications be implemented at least to the extent that all the required elements are present.

This specification prescribes the use and extension of previously established industry specification tables whenever possible. The trend to remove runtime call-based interfaces is well documented. The ACPI (Advanced Configuration and Power Interface) specification is an example of new and innovative firmware technologies that were designed on the premise that OS developers prefer to minimize runtime calls into firmware. ACPI focuses on no runtime calls to the BIOS.

Q.2.1 ACPI Specification

The interface defined by the Advanced Configuration and Power Interface (ACPI) Specification is the primary OS runtime interface for IA-32, x64 and Itanium platforms. ACPI fully defines the methodology that allows the OS to discover and configure all platform resources. ACPI allows the description of non-Plug and Play motherboard devices in a plug and play manner. ACPI also is capable of describing power management and hot plug events to the OS. (For more information on ACPI, see “Links to UEFI-Related Documents” (http://uefi.org/uefi) under the heading "ACPI"; see also http://uefi.org/acpi).

Q.2.2 Additional Considerations for Itanium-Based Platforms

Any information or service that is available in Itanium architecture firmware specifications supercedes any requirement in the common supported 32-bit and Itanium architecture specifications listed above. The Itanium architecture firmware specifications (currently the Itanium® System Abstraction Layer Specification and portions of the Intel® Itanium® Architecture Software Developer’s Manual, volumes 1–3) define the baseline functionality required for all Itanium architecture platforms. The major addition that UEFI makes to these Itanium architecture firmware specifications is that it defines a boot infrastructure and a set of services that constitute a common platform definition for high-volume Itanium architecture–based systems to implement based on the more generalized Itanium architecture firmware specifications.

The following specifications are the required Intel Itanium architecture specifications for all Itanium architecture–based platforms:

Appendix R - Glossary

_ADR
A reserved name in ACPI name space. It refers to an address on a bus that has standard enumeration. An example would be PCI, where the enumeration method is described in the PCI Local Bus specification.

_CRS
A reserved name in ACPI name space. It refers to the current resource setting of a device. A _CRS is required for devices that are not enumerated in a standard fashion. _CRS is how ACPI converts nonstandard devices into Plug and Play devices.

_HID
A reserved name in ACPI name space. It represents a device’s plug and play hardware ID and is stored as a 32-bit compressed EISA ID. _HID objects are optional in ACPI. However, a _HID object must be used to describe any device that will be enumerated by the ACPI driver in the OS. This is how ACPI deals with non–Plug and Play devices.

_UID
A reserved name in ACPI name space. It is a serial number style ID that does not change across reboots. If a system contains more than one device that reports the same _HID, each device must have a unique _UID. The _UID only needs to be unique for device that have the exact same _HID value.

ACPI Device Path
A Device Path that is used to describe devices whose enumeration is not described in an industry-standard fashion. These devices must be described using ACPI AML in the ACPI name space; this type of node provides linkage to the ACPI name space.

ACPI
Refers to the Advanced Configuration and Power Interface Specification and to the concepts and technology it discusses. The specification defines a new interface to the system board that enables the operating system to implement operating system-directed power management and system configuration.

Alt-GR Unicode
Represents the character code of a key when the Alt-GR modifier key is held down. This key (A2) in some keyboard layouts is defined as the right alternate key and serves the same function as the left alternate key. However, in many other layouts it is a secondary modifier key similar to shift. For instance, key C1 is equated to the letter a and its Unicode character code in the typical U.K. keyboard is a non-shifted character code of 0x0061. When holding down the Alt-GR key in conjunction with the pressing of key C1, , the value on the same keyboard often produces an á, which is a character code 0x00E1.
**Base Code (BC)**

The PXE Base Code, included as a core protocol in EFI, is comprised of a simple network stack (UDP/IP) and a few common network protocols (DHCP, Bootserver Discovery, TFTP) that are useful for remote booting machines.

**BC**

See Base Code (BC)

**Big Endian**

A memory architecture in which the low-order byte of a multibyte datum is at the highest address, while the high-order byte is at the lowest address. See Little Endian.

**BIOS Boot Specification Device Path**

A Device Path that is used to point to boot legacy operating systems; it is based on the BIOS Boot Specification, Version 1.01.

**BIOS Parameter Block (BPB)**

The first block (sector) of a partition. It defines the type and location of the FAT File System on a drive.

**BIOS**


**Block I/O Protocol**

A protocol that is used during boot services to abstract mass storage devices. It allows boot services code to perform block I/O without knowing the type of a device or its controller.

**Block Size**

The fundamental allocation unit for devices that support the Block I/O Protocol. Not less than 512 bytes. This is commonly referred to as sector size on hard disk drives.

**Boot Device**

The Device Handle that corresponds to the device from which the currently executing image was loaded.

**Boot Manager**

The part of the firmware implementation that is responsible for implementing system boot policy. Although a particular boot manager implementation is not specified in this document, such code is generally expected to be able to enumerate and handle transfers of control to the available OS loaders as well as UEFI applications and drivers on a given system. The boot manager would typically be responsible for interacting with the system user, where applicable, to determine what to load during system startup. In cases where user interaction is not indicated, the boot manager would determine what to load and, if multiple items are to be loaded, what the sequencing of such loads would be.
**Block Size**

The fundamental allocation unit for devices that support the Block I/O Protocol. Not less than 512 bytes. This is commonly referred to as sector size on disk drives.

**Boot Services Table**

A table that contains the firmware entry points for accessing boot services functions such as [Task Priority Services](#) and [Memory Allocation Services](#). The table is accessed through a pointer in the [System Table](#).

**Boot Services Time**

The period of time between platform initialization and the call to `ExitBootServices()`. During this time, [UEFI Driver](#) and applications are loaded iteratively and the system boots from an ordered list of EFI OS loaders.

**Boot Services**

The collection of interfaces and protocols that are present in the boot environment. The services minimally provide an OS loader with access to platform capabilities required to complete OS boot. Services are also available to drivers and applications that need access to platform capability. Boot services are terminated once the operating system takes control of the platform.

**BPB**

See [BIOS Parameter Block (BPB)](#).

**BTT**

Block Translation Table: A software mechanism for adding single block write atomicity to any Block Mode ranges or byte-addressable Persistent Memory ranges.

**Callback**

Target function which augments the Forms Processor’s ability to evaluate or process configuration settings. Callbacks are not available when the Forms Processor is operating in a Disconnected state.

**CIM**

See [Common Information Model (CIM)](#).

**Cluster**

A collection of disk sectors. Clusters are the basic storage units for disk files. See [File Allocation Table (FAT)](#).

**COFF**

Common Object File Format, a standard file format for binary images.

**Coherency Domain**

1. The global set of resources that is visible to at least one processor in a platform.
2. The address resources of a system as seen by a processor. It consists of both system memory and I/O space.
Common Information Model (CIM)
An object-oriented schema defined by the DMTF. CIM is an information model that provides a common way to describe and share management information enterprise-wide.

Console I/O Protocol
A protocol that is used during Boot Services to handle input and output of text-based information intended for the system administrator. It has two parts, a Simple Input Protocol that is used to obtain input from the ConsoleIn device and a Simple Text Output Protocol that is used to control text-based output devices. The Console I/O Protocol is also known as the EFI Console I/O Protocol.

ConsoleIn
The device handle that corresponds to the device used for user input in the boot services environment. Typically the system keyboard.

ConsoleOut
The device handle that corresponds to the device used to display messages to the user from the boot services environment. Typically a display screen.

DBCS
Double Byte Character Set.

Desktop Management Interface (DMI)
A platform management information framework, built by the DMTF and designed to provide manageability for desktop and server computing platforms by providing an interface that is:
1) independent of any specific desktop operating system, network operating system, network protocol, management protocol, processor, or hardware platform;
2) easy for vendors to implement; and
3) easily mapped to higher-level protocols.

DMTF (formerly Desktop Management Task Force)
The DMTF is a standards organization comprised of companies from all areas of the computer industry. DMTF creates open manageability standards spanning diverse emerging and traditional IT infrastructures including cloud, virtualization, network, servers and storage.

Device Handle
A handle points to a list of one or more protocols that can respond to requests for services for a given device referred to by the handle.

Device I/O Protocol
A protocol that is used during boot services to access memory and I/O. Also called the EFI Device I/O Protocol.

Device Path Instance
When an environment variable represents multiple devices, it is possible for a device path to contain multiple device paths. An example of this would be the ConsoleOut.
environment variable that consists of both a VGA console and a serial output console. This environment variable would describe a console output stream that would send output to both devices and therefore has a Device Path that consists of two complete device paths. Each of these paths is a device path instance.

**Device Path Node**
A variable-length generic data structure that is used to build a device path. Nodes are distinguished by type, subtype, length, and path-specific data. See [Device Path](#).

**Device Path Protocol**
A protocol that is used during boot services to provide the information needed to construct and manage Device Paths. Also called the EFI Device Path Protocol.

**Device Path**
A variable-length binary data structure that is composed of variable-length generic device path nodes and is used to define the programmatic path to a logical or physical device. There are six major types of device paths: [Hardware Device Path](#), [ACPI Device Path](#), [Messaging Device Path](#), [Media Device Path](#), [BIOS Boot Specification Device Path](#), and [End of Hardware Device Path](#).

**DHCP**
See [Dynamic Host Configuration Protocol (DHCP)](#).

**Disconnected**
The state when a Forms Processor is manipulating a form set without being connected to the Target’s pre-OS environment. For example, after booting an OS, a Forms Processor cannot execute call-backs or read the configuration settings. For example, when running a Forms Browser while on a remote machine that is not connected to the Target. In these cases, the Forms Processor has limited knowledge of the Target’s current configuration settings and limited or no ability to use call-backs.

**Disk I/O Protocol**
A protocol that is used during boot services to abstract Block I/O devices to allow non-block-sized I/O operations. Also called the EFI Disk I/O Protocol.

**DMI**
See [DBCS](#).

**DMTF**
See [DMTF (formerly Desktop Management Task Force)](#).

**DNS**
Domain Name System. A protocol used manipulating and translating hostname and IP address

**DTLS**
Datagram Transport Layer Security. A protocol to provide communication privacy above UDP.
**Dynamic Host Configuration Protocol (DHCP)**
A protocol that is used to get information from a configuration server. DHCP is defined by the DMTF (formerly Desktop Management Task Force), not EFI.

**EAP**

**EBC Image**
Executable EBC image following the PE32 file format.

**EBC**
See EFI Byte Code (EBC).

**EFI**
Extensible Firmware Interface. An interface between the operating system (OS) and the platform firmware.

**EFI Byte Code (EBC)**
The binary encoding of instructions as output by the EBC C compiler and linker. The EBC Image is executed by the interpreter.

**EFI File**
A container consisting of a number of blocks that holds an image or a data file within a file system that complies with this specification.

**EFI Hard Disk**
A hard disk that supports the new EFI partitioning scheme (GUID Partition).

**EFI-compliant**
Refers to a platform that complies with this specification.

**EFI-conformant**
See EFI-compliant.

**End of Hardware Device Path**
A Device Path which, depending on the subtype, is used to indicate the end of the Device Path instance or Device Path structure.

**Enhanced Mode (EM)**
The 64-bit architecture extension that makes up part of the Intel® Itanium® architecture.

**Event Services**
The set of functions used to manage events. Includes EFI_BOOT_SERVICES.CheckEvent(), EFI_BOOT_SERVICES.CreateEvent(), EFI_BOOT_SERVICES.CloseEvent(), EFI_BOOT_SERVICES.SignalEvent(), and EFI_BOOT_SERVICES.WaitForEvent().
Event
An EFI data structure that describes an “event”—for example, the expiration of a timer.

Event Services
The set of functions used to manage events. Includes EFI_BOOT_SERVICES.CheckEvent(), EFI_BOOT_SERVICES.CreateEvent(), EFI_BOOT_SERVICES.CloseEvent(), EFI_BOOT_SERVICES.SignalEvent(), and EFI_BOOT_SERVICES.WaitForEvent().

FAT File System
The file system on which the EFI File system is based. See File Allocation Table (FAT) and GUID Partition Table (GPT).

FAT
See File Allocation Table (FAT).

File Allocation Table (FAT)
A table that is used to identify the clusters that make up a disk file. File allocation tables come in three flavors: FAT12, which uses 12 bits for cluster numbers; FAT16, which uses 16 bits; and FAT32, which allots 32 bits but only uses 28 (the other 4 bits are reserved for future use).

File Handle Protocol
A component of the File System Protocol. It provides access to a file or directory. Also called the EFI File Handle Protocol.

File System Protocol
A protocol that is used during boot services to obtain file-based access to a device. It has two parts, a Simple File System Protocol that provides a minimal interface for file-type access to a device, and a File Handle Protocol that provides access to a file or directory.

Firmware
Any software that is included in read-only memory (ROM).

Font
A graphical representation corresponding to a character set, in this case Unicode. The following are the same Latin letter in three fonts using the same size (14):

A
A
A

Font glyph
The individual elements of a font corresponding to single characters are called font glyphs or simply glyphs. The first character in each of the above three lines is a glyph for the letter "A" in three different fonts.
Form
Logical grouping of questions with a unique identifier.

Form Set
An HII database package describing a group of forms, including one parent form and zero or more child forms.

Forms Browser
A Forms Processor capable of displaying the user-interface information a display and interacting with a user.

Forms Processor
An application capable of reading and processing the forms data within a forms set.

Globally Unique Identifier (GUID)
A 128-bit value used to differentiate services and structures in the boot services environment. The format of a GUID is defined in Appendix A. See Protocol.

Glyph
The individual elements of a font corresponding to single characters. May also be called font keyboard layout glyphs. Also see font glyph above.

GPT: See GUID Partition Table (GPT).

GPT disk layout:
The data layout on a disk consisting of a protective MBR in LBA 0, a GPT Header in LBA 1, and additional GPT structures and partitions in the remaining LBAs. See chapter 5.

GPTHeader
The header in a GUID Partition Table (GPT). Among other things, it contains the number of GPT Partition Entries and the first and last LBAs that can be used for the entries.

GPT Partition Entry
A data structure that characterizes a Partition in the GPT disk layout. Among other things, it specifies the starting and ending LBA of the partition.

GUID Partition Table (GPT)
A data structure that describes one or more partitions. It consists of a GPTHeader and, typically, at least one GPTPartition Entry. There are two GUID partition tables: the Primary Partition Table (located in LBA 1 of the disk) and a Backup Partition Table (located in the last LBA of the disk). The Backup Partition Table is a copy of the Primary Partition Table.

GPTPartition Entry
A data structure that characterizes a GUID Partition. Among other things, it specifies the starting and ending LBA of the partition.
GUID Partition
A contiguous group of sectors on an EFI Hard Disk.

Handle
See Device Handle.

Hardware Device Path
A Device Path that defines how a hardware device is attached to the resource domain of a system (the resource domain is simply the shared memory, memory mapped I/O, and I/O space of the system).

HII
Human Interface Infrastructure.

HII Database
The centralized repository for HII-related information, organized as package lists.

HTML
Hypertext Markup Language. A particular implementation of SGML focused on hypertext applications. HTML is a fairly simple language that enables the description of pages (generally Internet pages) that include links to other pages and other data types (such as graphics). When applied to a larger world, HTML has many shortcomings, including localization (q.v.) and formatting issues. The HTML form concept is of particular interest to this application.

HTTP
Hypertext transfer protocol. HTTP functions as request-response protocol in the client-server computing rule.

IA-32
See Intel® Architecture-32 (IA-32).

IFR
Internal Form Representation. Used to represent forms in EFI so that it can be interpreted as is or expanded easily into XHTML.

Image Handle
A handle for a loaded image; image handles support the loaded image protocol.

Image Handoff State
The information handed off to a loaded image as it begins execution; it consists of the image’s handle and a pointer to the image’s system table.

Image Header
The initial set of bytes in a loaded image. They define the image’s encoding.

Image Services
The set of functions used to manage EFI images. Includes EFI_BOOT_SERVICES.LoadImage(), EFI_BOOT_SERVICES.StartImage(),
 EFI_BOOT_SERVICES.UnloadImage(), EFI_BOOT_SERVICES.Exit(),
EFI_BOOT_SERVICES.ExitBootServices(), and EFI_IMAGE_ENTRY_POINT.

**Image**

(1) An executable file stored in a file system that complies with this specification. Images may be drivers, applications or OS loaders. Also called an EFI Image.

(2) Executable binary file containing EBC and data. Output by the EBC linker.

**IME**

Input Method Editor. A program or subprogram that is used to map keystrokes to logographic characters. For example, IMEs are used (possibly with user intervention) to map the Kana (Hirigana or Katakana) characters on Japanese keyboards to Kanji.

**Intel® Architecture-32 (IA-32)**


**Intel® Itanium® Architecture**

The Intel architecture that has 64-bit instruction capabilities, new performance-enhancing features, and support for the IA-32 instruction set. This architecture is described in the *Itanium™ Architecture Software Developer’s Manual*.

**Internationalization**

In this context, is the process of making a system usable across languages and cultures by using universally understood symbols. Internationalization is difficult due to the differences in cultures and the difficulty of creating obvious symbols; for example, why does a red octagon mean "Stop"?

**Interpreter**

The software implementation that decodes EBC binary instructions and executes them on a VM. Also called EBC interpreter.

**Keyboard layout**

The physical representation of a user’s keyboard. The usage of this is in conjunction to a structure that equates the physical key(s) and the associated action it represents. For instance, key C1 is equated to the letter a and its Unicode value in the typical U.K. keyboard is a non-shifted value of 0x0061.

**LAN On Motherboard (LOM)**

This is a network device that is built onto the motherboard (or baseboard) of the machine.

**LBA:**

See Logical Block Address (LBA).
**Legacy Platform**
A platform which, in the interests of providing backward-compatibility, retains obsolete technology.

**LFN**
See Long File Names (LFN).

**Little Endian**
A memory architecture in which the low-order byte of a multibyte datum is at the lowest address, while the high-order byte is at the highest address. See Big Endian.

**Load File Protocol**
A protocol that is used during boot services to find and load other modules of code.

**Loaded Image Protocol**
A protocol that is used during boot services to obtain information about a loaded image. Also called the EFI Loaded Image Protocol.

**Loaded Image**
A file containing executable code. When started, a loaded image is given its image handle and can use it to obtain relevant image data.

**Localization**
The process of focusing a system in so that it works using the symbols of a language/culture. To a major extent the following design is influenced by the requirements of localization.

**Logical Block Address (LBA):**
The address of a logical block on a disk. The first LBA on a disk is LBA 0.

**Logographic**
A character set that uses characters to represent words or parts of words rather than syllables or sounds. Kanji is logographic but Kana characters are not.

**LOM**
See LAN On Motherboard (LOM).

**Long File Names (LFN)**
Refers to an extension to the FAT File System that allows file names to be longer than the original standard (eight characters plus a three-character extension).

**Machine Check Abort (MCA)**
The system management and error correction facilities built into the Intel Itanium processors.

**Master Boot Record (MBR)**
The data structure that resides on the LBA 0 of a hard disk and defines the partitions on the disk.
MBR

See Master Boot Record (MBR).

MBR boot code:

x86 code in the first LBA of a disk.

MBR disk layout:

The data layout on a disk consisting of an MBR in LBA 0 and partitions described by the MBR in the remaining LBAs. See chapter 5 and Appendix NEW.

MBR Partition Record

A data structure that characterizes a Partition in the MBR disk layout.

MCA

See Machine Check Abort (MCA).

Media Device Path

A Device Path that is used to describe the portion of a medium that is being abstracted by a boot service. For example, a Media Device Path could define which partition on a hard drive was being used.

Memory Allocation Services

The set of functions used to allocate and free memory, and to retrieve the memory map. Includes `EFI_BOOT_SERVICES.AllocatePages()`, `EFI_BOOT_SERVICES.FreePages()`, `EFI_BOOT_SERVICES.AllocatePool()`, `EFI_BOOT_SERVICES.FreePool()`, and `EFI_BOOT_SERVICES.GetMemoryMap()`.

Memory Map

A collection of structures that defines the layout and allocation of system memory during the boot process. Drivers and applications that run during the boot process prior to OS control may require memory. The boot services implementation is required to ensure that an appropriate representation of available and allocated memory is communicated to the OS as part of the hand-off of control.

Memory Type

One of the memory types defined by UEFI for use by the firmware and UEFI applications. Among others, there are types for boot services code, boot services data, Runtime Services code, and runtime services data. Some of the types are used for one purpose before `EFI_BOOT_SERVICES.ExitBootServices()` is called and another purpose after.

Messaging Device Path

A Device Path that is used to describe the connection of devices outside the Coherency Domain of the system. This type of node can describe physical messaging information (e.g., a SCSI ID) or abstract information (e.g., networking protocol IP addresses).
Miscellaneous Service

Various functions that are needed to support the EFI environment. Includes EFI_BOOT_SERVICES.InstallConfigurationTable(), ResetSystem(), EFI_BOOT_SERVICES.Stall(), EFI_BOOT_SERVICES.SetWatchdogTimer(), EFI_BOOT_SERVICES.GetNextMonotonicCount(), and EFI_BOOT_SERVICES.GetNextHighMonotonicCount().

MTFTP

See Multicast Trivial File Transfer Protocol (MTFTP).

Multicast Trivial File Transfer Protocol (MTFTP)

A protocol used to download a Network Boot Program to many clients simultaneously from a TFTP server.

Name Space or Namespace

A namespace defines a contiguously-addressed range of Non-Volatile Memory conceptually similar to a SCSI Logical Unit (LUN) or a NVM Express namespace. In general, a collection of device paths; in an EFI Device Path.

Native Code

Low level instructions that are native to the host processor. As such, the processor executes them directly with no overhead of interpretation. Contrast this with EBC, which must be interpreted by native code to operate on a VM.

NBP

See Network Bootstrap Program (NBP) or Network Boot Program.

Network Boot Program

A remote boot image downloaded by a PXE client using the Trivial File Transport Protocol (TFTP) or the Multicast Trivial File Transfer Protocol (MTFTP). See Network Bootstrap Program (NBP).

Network Bootstrap Program (NBP)

This is the first program that is downloaded into a machine that has selected a PXE capable device for remote boot services.

A typical NBP examines the machine it is running on to try to determine if the machine is capable of running the next layer (OS or application). If the machine is not capable of running the next layer, control is returned to the EFI boot manager and the next boot device is selected. If the machine is capable, the next layer is downloaded and control can then be passed to the downloaded program.

Though most NBPs are OS loaders, NBPs can be written to be standalone applications such as diagnostics, backup/restore, remote management agents, browsers, etc.

Network Interface Card (NIC)

Technically, this is a network device that is inserted into a bus on the motherboard or in an expansion board. For the purposes of this document, the term NIC will be used
in a generic sense, meaning any device that enables a network connection (including LOMs and network devices on external buses (USB, 1394, etc.)).

**NIC**

See [Network Interface Card (NIC)](#).

**Non-spacing key**

Typically an accent key that does not advance the cursor and is used to create special characters similar to ÄäÊê. This function is provided only on certain keyboard layouts.

**NV**

Nonvolatile.

**Package**

HII information with a unique type, such as strings, fonts, images or forms.

**Package List**

Group of packages identified by a GUID.

**Page Memory**

A set of contiguous pages. Page memory is allocated by `EFI_BOOT_SERVICES.AllocatePages()` and returned by ` EFI_BOOT_SERVICES.FreePages()`.

**Partition Discovery**

The process of scanning a block device to determine whether it contains a Partition.

**Partition**

A contiguous set of LBAs on a disk, described by the MBR and/or GPT disk layouts.

**PC-AT**

Refers to a PC platform that uses the AT form factor for their motherboards.

**PCI Bus Driver**

Software that creates a handle for every PCI Controller on a PCI Host Bus Controller and installs both the PCI I/O Protocol and the Device Path Protocol onto that handle. It may optionally perform PCI Enumeration if resources have not already been allocated to all the PCI Controllers on a PCI Host Bus Controller. It also loads and starts any UEFI drivers found in any PCI Option ROMs discovered during PCI Enumeration. If a driver is found in a PCI Option ROM, the PCI Bus Driver will also attach the Bus Specific Driver Override Protocol to the handle for the PCI Controller that is associated with the PCI Option ROM that the driver was loaded from.

**PCI Bus**

A collection of up to 32 physical PCI Devices that share the same physical PCI bus. All devices on a PCI Bus share the same PCI Configuration Space.
**PCI Configuration Space**

The configuration channel defined by PCI to configure **PCI Devices** into the resource domain of the system. Each PCI device must produce a standard set of registers in the form of a PCI Configuration Header, and can optionally produce device specific registers. The registers are addressed via Type 0 or Type 1 PCI Configuration Cycles as described by the **PCI Specification**. The PCI Configuration Space can be shared across multiple **PCI Buses**. On most **PC-AT** architecture systems and typical Intel® chipsets, the PCI Configuration Space is accessed via I/O ports 0xCF8 and 0xCFC. Many other implementations are possible.

**PCI Controller**

A hardware component that is discovered by a **PCI Bus Driver**, and is managed by a **PCI Device Driver, PCI Functions** and **PCI Controller** are used equivalently in this document.

**PCI Device Driver**

Software that manages one or more PCI Controllers of a specific type. A driver will use the **PCI I/O Protocol** to produce a device I/O abstraction in the form of another protocol (i.e., Block I/O, Simple Network, Simple Input, Simple Text Output, Serial I/O, Load File).

**PCI Devices**

A collection of up to 8 **PCI Functions** that share the same **PCI Configuration Space**. A PCI Device is physically connected to a **PCI Buses**.

**PCI Enumeration**

The process of assigning resources to all the PCI Controllers on a given **PCI Host Bus Controller**. This includes PCI Bus Number assignments, PCI Interrupt assignments, PCI I/O resource allocation, the PCI Memory resource allocation, the PCI Prefetchable Memory resource allocation, and miscellaneous PCI DMA settings.

**PCI Functions**

A controller that provides some type of I/O services. It consumes some combination of PCI I/O, PCI Memory, and PCI Prefetchable Memory regions, and up to 256 bytes of the **PCI Configuration Space**. The PCI Function is the basic unit of configuration for PCI.

**PCI Host Bus Controller**

A chipset component that produces PCI I/O, PCI Memory, and PCI Prefetchable Memory regions in a single Coherency Domain. A PCI Host Bus Controller is composed of one or more **PCI Root Bridges**.

**PCI I/O Protocol**

A software interface that provides access to PCI Memory, PCI I/O, and PCI Configuration spaces for a PCI Controller. It also provides an abstraction for PCI Bus Master DMA.
PCI Option ROM
A ROM device that is accessed through a PCI Controller, and is described in the PCI Controller’s Configuration Header. It may contain one or more PCI Device Drivers that are used to manage the PCI Controller.

PCI Root Bridge I/O Protocol
A software abstraction that provides access to the PCI I/O, PCI Memory, and PCI Prefetchable Memory regions in a single Coherency Domain.

PCI Root Bridge
A chipset component(s) that produces a physical PCI Local Bus.

PCI Segment
A collection of up to 256 PCI Buses that share the same PCI Configuration Space. PCI Segment is defined in the ACPI Specification as the _SEG object. The SAL_PCI_CONFIG_READ and SAL_PCI_CONFIG_WRITE procedures defined in chapter 9 of the SAL Specification define how to access the PCI Configuration Space in a system that supports multiple PCI Segments. If a system only supports a single PCI Segment the PCI Segment number is defined to be zero. The existence of PCI Segments enables the construction of systems with greater than 256 PCI buses.

Pool Memory
A set of contiguous bytes. A pool begins on, but need not end on, an “8-byte” boundary. Pool memory is allocated in pages—that is, firmware allocates enough contiguous pages to contain the number of bytes specified in the allocation request. Hence, a pool can be contained within a single page or extend across multiple pages. Pool memory is allocated by EFI_BOOT_SERVICES.AllocatePool() and returned by EFI_BOOT_SERVICES.FreePool().

Preboot Execution Environment (PXE)
A means by which agents can be loaded remotely onto systems to perform management tasks in the absence of a running OS. To enable the interoperability of clients and downloaded bootstrap programs, the client preboot code must provide a set of services for use by a downloaded bootstrap. It also must ensure certain aspects of the client state at the point in time when the bootstrap begins executing.

The complete PXE specification covers three areas; the client, the network and the server.

Client
• Makes network devices into bootable devices.
• Provides APIs for PXE protocol modules in EFI and for universal drivers in the OS.

Network
• Uses existing technology: DHCP, TFTP, etc.
• Adds “vendor-specific” tags to DHCP to define PXE-specific operation within DHCP.
- Adds multicast TFTP for high bandwidth remote boot applications.
- Defines Bootserver discovery based on DHCP packet format.

**Server**

- **Bootserver**: Responds to Bootserver discovery requests and serves up remote boot images.

- **proxyDHCP**: Used to ease the transition of PXE clients and servers into existing network infrastructure. proxyDHCP provides the additional DHCP information that is needed by PXE clients and Bootservers without making changes to existing DHCP servers.

- **MTFTP**: Adds multicast support to a TFTP server.

- **Plug-In Modules**: Example proxyDHCP and Bootservers provided in the PXE SDK (software development kit) have the ability to take plug-in modules (PIMs). These PIMs are used to change/enhance the capabilities of the proxyDHCP and Bootservers.

**Protocol Handler Services**

The set of functions used to manipulate handles, protocols, and protocol interfaces. Includes `EFI_BOOT_SERVICES.InstallProtocolInterface()`, `EFI_BOOT_SERVICES.UninstallProtocolInterface()`, `EFI_BOOT_SERVICES.ReinstallProtocolInterface()`, `EFI_BOOT_SERVICES.HandleProtocol()`, `EFI_BOOT_SERVICES.RegisterProtocolNotify()`, `EFI_BOOT_SERVICES.LocateHandle()`, and `EFI_BOOT_SERVICES.LocateDevicePath()`.

**Protocol Handler**

A function that responds to a call to a `HandleProtocol` request for a given handle. A protocol handler returns a protocol interface structure.

**Protocol Interface Structure**

The set of data definitions and functions used to access a particular type of device. For example, `BLOCK_IO` is a protocol that encompasses interfaces to read and write blocks from mass storage devices. See **Protocol**.

**Protocol Revision Number**

The revision number associated with a protocol. See **Protocol**.

**Protocol**

The information that defines how to access a certain type of device during boot services. A protocol consists of a **Globally Unique Identifier (GUID)**, a protocol revision number, and a protocol interface structure. The interface structure contains data definitions and a set of functions for accessing the device. A device can have multiple protocols. Each protocol is accessible through the device’s handle.
PXE Base Code Protocol
A protocol that is used to control PXE-compatible devices. It may be used by the firmware’s boot manager to support booting from remote locations. Also called the EFI PXE Base Code Protocol.

PXE
See Preboot Execution Environment (PXE).

Question
IFR which describes how a single configuration setting should be presented, stored, and validated.

Read-Only Memory (ROM)
When used with reference to the UNDI specification, ROM refers to a nonvolatile memory storage device on a NIC.

Reset
The action which forces question values to be reset to their defaults.

ROM
See Question.

Runtime Services Table
A table that contains the firmware entry points for accessing runtime services functions such as Time Services and Virtual Memory Services. The table is accessed through a pointer in the System Table.

Runtime Services
Interfaces that provide access to underlying platform specific hardware that may be useful during OS runtime, such as timers. These services are available during the boot process but also persist after the OS loader terminates boot services.

SAL
See System Abstraction Layer (SAL).

scan code
A value representing the location of a key on a keyboard. Scan codes may also encode make (key press) and break (key release) and auto-repeat information.

Serial Protocol
A Protocol that is used during boot services to abstract byte stream devices—that is, to communicate with character-based I/O devices.

SGML
shifted Unicode

Shifted Unicode represents the Unicode character code of a key when the shift modifier key is held down. For instance, key C1 is equated to the letter a and its Unicode character code in the typical U.K. keyboard is a non-shifted value of 0x0061. When the shift key is held down in conjunction with the pressing of key C1, however, the value on the same keyboard often produces an A, which is a the Unicode character code 0x0041.

Protocol that is used during boot services to abstract byte stream devices—that is, to communicate with character-based I/O devices.

Simple File System Protocol

A component of the File System Protocol. It provides a minimal interface for file-type access to a device.

Simple Input Protocol

A protocol that is used to obtain input from the ConsoleIn device. It is one of two protocols that make up the Console I/O Protocol.

Simple Network Protocol

A protocol that is used to provide a packet-level interface to a network adapter. Also called the EFI Simple Network Protocol.

Simple Text Output Protocol

A protocol that is used to control text-based output devices. It is one of two protocols that make up the Console I/O Protocol.

SKU

Stock keeping unit. An acronym commonly used to reference a “version” of a particular platform. An example might be “We have three different SKUs of this platform.”

SMBIOS

See System Management BIOS (SMBIOS).

SNIA

Storage Network Industry Association.(www.snia.org)

SNIA Common RAID Disk Data Format


SSL

Secure Sockets Layer. A security protocol that provides communications privacy over the Internet. It is predecessor to TLS.
**StandardError**

The device handle that corresponds to the device used to display error messages to the user from the boot services environment.

**Status Codes**

Success, error, and warning codes returned by boot services and runtime services functions.

**string**

A null-terminated array of 16-bit UCS-2 encoded Unicode characters. All strings in this specification are encoded using UCS-2 unless otherwise specified.

**Submit**

The action which forces modified question values to be written back to storage.

**System Abstraction Layer (SAL)**

Firmware that abstracts platform implementation differences, and provides the basic platform software interface to all higher level software.

**System Management BIOS (SMBIOS)**

A table-based interface that is required by the *Wired for Management Baseline Specification*. It is used to relate platform-specific management information to the OS or to an OS-based management agent.

**System Table**

Table that contains the standard input and output handles for a UEFI application, as well as pointers to the boot services and runtime services tables. It may also contain pointers to other standard tables such as the ACPI, SMBIOS, and SAL System tables. A loaded image receives a pointer to its system table when it begins execution. Also called the EFI System Table.

**Target**

The system being configured.

**Task Priority Level (TPL)**

The boot services environment exposes three task priority levels: “normal,” “callback,” and “notify.”

**Task Priority Services**

The set of functions used to manipulate task priority levels. Includes EFI_BOOT_SERVICES.RaiseTPL() and EFI_BOOT_SERVICES.RestoreTPL().

**TFTP**

See *Trivial File Transport Protocol (TFTP)*.

**Time Format**

The format for expressing time in an EFI-compliant platform. For more information, see Appendix A.
Time Services
The set of functions used to manage time. Includes `GetTime()`, `SetTime()`, `GetWakeupTime()`, and `SetWakeupTime()`.

Timer Services
The set of functions used to manipulate timers. Contains a single function, `EFI_BOOT_SERVICES.SetTimer()`.

TLS
Transport Layer Security. A protocol to provide privacy and data integrity between two communicating applications above TCP.

TPL
See Target.

Trivial File Transport Protocol (TFTP)
A protocol used to download a Network Boot Program from a TFTP server.

UEFI
Unified Extensible Firmware Interface. The interface between the operating system (OS) and the platform firmware defined by this specification.

UEFI Application
Modular code that may be loaded in the boot services environment to accomplish platform specific tasks within that environment. Examples of possible applications might include diagnostics or disaster recovery tools shipped with a platform that run outside the OS environment. UEFI applications may be loaded in accordance with policy implemented by the platform firmware to accomplish a specific task. Control is then returned from the UEFI application to the platform firmware.

UEFI Boot Service Driver
A UEFI driver that is loaded into boot services memory and stays resident until boot services terminate.

UEFI Driver
A module of code typically inserted into the firmware via protocol interfaces. Drivers may provide device support during the boot process or they may provide platform services. It is important not to confuse UEFI drivers with OS drivers that load to provide device support once the OS takes control of the platform.

UEFI OS Loader
A UEFI application that is the first piece of operating system code loaded by the firmware to initiate the OS boot process. This code is loaded at a fixed address and then executed. The OS takes control of the system prior to completing the OS boot process by calling the interface that terminates all boot services.

UEFI Runtime Services Driver
A UEFI driver that is loaded into runtime services memory and stays resident during runtime.
**Unaccepted Memory**

Some Virtual Machine platforms, such as AMD SEV-SNP, introduce the concept of memory acceptance, requiring memory to be accepted before it can be used by the guest. This protects against a class of attacks by the virtual machine platform.

**UNDI**

See [Universal Network Device Interface (UNDI)](#).

**Unicode Collation Protocol**

A protocol that is used during boot services to perform case-insensitive comparisons of strings.

**Unicode**

An industry standard internationalized character set used for human readable message display.

**Universal Network Device Interface (UNDI)**

UNDI is an architectural interface to NICs. Traditionally NICs have had custom interfaces and custom drivers (each NIC had a driver for each OS on each platform architecture). Two variations of UNDI are defined in this specification: H/W UNDI and S/W UNDI. H/W UNDI is an architectural hardware interface to a NIC. S/W UNDI is a software implementation of the H/W UNDI.

**Universal Serial Bus (USB)**

A bi-directional, isochronous, dynamically attachable serial interface for adding peripheral devices such as serial ports, parallel ports, and input devices on a single bus.

**URI**

Uniform resource identifier. URI is a string of characters used to identify a name of a resource.

**USB Bus Driver**

Software that enumerates and creates a handle for each newly attached USB Controller and installs both the USB I/O Protocol and the Device Path Protocol onto that handle, starts that device driver if applicable. For each newly detached USB Controller, the device driver is stopped, the USB I/O Protocol and the Device Path Protocol are uninstalled from the device handle, and the device handle is destroyed.

**USB Bus**

A collection of up to 127 physical USB Devices that share the same physical USB bus. All devices on a USB Bus share the bandwidth of the USB Bus.

**USB Controller**

A hardware component that is discovered by a USB Bus Driver, and is managed by a USB Device Driver. USB Interface and USB Controller are used equivalently in this document.
**USB Device Driver**
Software that manages one or more USB Controller of a specific type. A driver will use the USB I/O Protocol to produce a device I/O abstraction in the form of another protocol (i.e., Block I/O, Simple Network, Simple Input, Simple Text Output, Serial I/O, Load File).

**USB Device**
A USB peripheral that is physically attached to the USB Bus.

**USB Enumeration**
A periodical process to search the USB Bus to detect if there have been any USB Controller attached or detached. If an attach event is detected, then the USB Controllers device address is assigned, and a child handle is created. If a detach event is detected, then the child handle is destroyed.

**USB Host Controller**
Moves data between system memory and devices on the USB Bus by processing data structures and generating the USB transactions. For USB 1.1, there are currently two types of USB Host Controllers: UHCI and OHCI.

**USB Hub**
A special USB Device through which more USB devices can be attached to the USB Bus.

**USB I/O Protocol**
A software interface that provides services to manage a USB Controller, and services to move data between a USB Controller and system memory.

**USB Interface**
The USB Interface is the basic unit of a physical USB Device.

**USB**
See Universal Serial Bus (USB).

**Variable Services**
The set of functions used to manage variables. Includes GetVariable(), SetVariable(), and GetNextVariableName().

**Virtual Memory Services**
The set of functions used to manage virtual memory. Includes SetVirtualAddressMap() and ConvertPointer().

**VM**
The Virtual Machine, a pseudo processor implementation consisting of registers which are manipulated by the interpreter when executing EBC instructions.
Watchdog Time
An alarm timer that may be set to go off. This can be used to regain control in cases where a code path in the boot services environment fails to or is unable to return control by the expected path.

WfM
See Wired for Management (WfM).

Wired for Management (WfM)
Refers to the Wired for Management Baseline Specification. The Specification defines a baseline for system manageability issues; its intent is to help lower the cost of computer ownership.

x64
Processors that are compatible with instruction sets and operation modes as exemplified by the AMD64 or Intel® Extended Memory 64 Technology (Intel® EM64T) architecture.

XHTML
Extensible HTML. XHTML "will obey all of the grammar rules of XML (properly nested elements, quoted attributes, and so on), while conforming to the vocabulary of HTML (the elements and attributes that are available for use and their relationships to one another)." [PXML, pg., 153]. Although not completely defined, XHTML is basically the intersection of XML and HTML and does support forms.

XML
Extensible Markup Language. A subset of SGML. Addresses many of the problems with HTML but does not currently (1.0) support forms in any specified way.
Index

Symbols
!PXE structure field definitions 2203
!PXE structures 2202

Numerics
32/64-bit UNDI interface 2202

A
ACPI 2402
ACPI_ADR 332
ACPI_ADR Device Path 298
ACPI Device Path, definition of 2403
ACPI name space 2188, 2193
ACPI Source Language 291
ACPI Terms 2193
ACPI, definition of 2403
ADD 923
ADR, definition of 2403
Advanced Configuration and Power Interface specification 2402
Advanced Configuration and Power Interface specification See also related information
AllocateBuffer() 672, 713
AllocatePages() 165
AllocatePool() 173
alphabetic function lists 2337
AND 924
ANSI 3.64 terminals, and SIMPLE_TEXT_OUTPUT 2183
Application, EFI 16, 17
ARP cache entries 1049
ARP Protocol
Functions
Add() 1446
Configure() 1444
Delete() 1449
Find() 1448
Flush() 1450
Request() 1451, 1452
GUID 1443
Interface Structure 1443
ARP Service Binding Protocol
GUID 1442
Arp() 1068
Arrow shapes 453
ASHR 925
ASL See ACPI Source Language
AsyncInterruptTransfer() 790
AsyncIsochronousTransfer() 797
Attribute bits, EFI PCI I/O Protocol 696
Attribute bits, PCI Root Bridge I/O attributes
   architecturally defined 81
Attributes, SIMPLE_TEXT_OUTPUT 457
Attributes() 718

B
Base Code (BC), definition of 2404
Big Endian, definition of 2404
BIOS code 5
BIOS Parameter Block 505
BIOS Parameter Block (BPB), definition of 2404
BIOS, definition of 2404
BIS_ALG_ID 1095
BIS_APPLICATION_HANDLE 1085
BIS_CERT_ID 1094
Block Elements Code Chart 453
Block I/O Protocol 552, 560
   Functions
      FlushBlocks() 559, 565
      Readblocks() 556, 561
      Reset() 556, 560
      WriteBlocks() 558, 563
      GUID 552, 560
      Interface Structure 553, 560
      Revision Number 552
Block Size, definition of 2404
Blt buffer 484
Blt Operation Table 492, 496
Blt() 491
Boot Device, definition of 2404
Boot Integrity Services Protocol 1082
   Functions
      Free() 1088
      GetBootObjectAuthorizationCertificate() 1089
GetBootObjectAuthorizationCheckFlag() 1090
GetBootObjectAuthorizationUpdateToken() 1091
GetSignatureInfo() 1092
Initialize() 1084
Shutdown() 1087
UpdateBootObjectAuthorizationUpdateBootObjectAuthorization_EFI_BIS() 1096
VerifyBootObject() 1104
VerifyObjectWithCredential() 1110
GUID 1082
   Interface Structure 1082
boot manager 70
   default media boot 72
Boot Manager, definition of 2404
boot mechanisms 88
boot order list 70
boot process
   illustration of 14
   overview 14
boot sequence 70
Boot Services 143, 233
   global functions 143, 233
   handle-based functions 143, 233
boot services 7
Boot Services Driver, definition of 2423
Boot Services Table, definition of 2405
Boot Services Table, EFI 91
Boot Services Time, definition of 2405
Boot Services, definition of 2405
booting
   future boot media 90
   via a network device 90
   via Load File Protocol 89
   via Simple File Protocol 88
booting from
   CD-ROM and DVD-ROM 509
diskettes 509
   hard drives 509
   network devices 510
   removable media 509
BPB See BIOS Parameter Block
BREAK 926
BulkTransfer() 788
bus-specific driver override protocol 386
C
CalculateCrc32() 232
CALL 928
Callback() 1080
calling conventions 24
genral 19
IA-32 21
CDB 2207
CheckEvent() 159
_CID 296
ClearRootHubPortFeature() 805
ClearScreen() 458
Close() 516
CloseEvent() 156
CloseEventExCreateEventEx 152
CloseProtocol() 199
Cluster, definition of 2405
CMP 930
CMPI 931
COFF, definition of 2405
Coherency Domain, definition of 2405
Common Information Model (CIM), definition of 2406
compressed data
bit order 886
block body 889
block header 888
format 886, 887
overall structure 887
Compression Algorithm Specification 885
compression source code 2288
compressor design 890
Configuration() 678
ConnectController() 202
Console 2180
Console I/O protocol 436
ConsoleIn 436
ConsoleIn, definition of 2406
ConsoleOut 448
ConsoleOut, definition of 2406
ControlTransfer() 785
conventions 10
data structure descriptions 10
function descriptions 10
instruction descriptions 11
procedure descriptions 10
protocol descriptions 10
pseudo-code conventions 11
ConvertPointer() 266
CopyMem() 228, 668, 709
CreateEvent() 148
CreateEventEx 144, 152, 153, 158
CreateThunk() 965
_CRS, definition of 2403
cursor movement 2284

D
Debug Image Info Table 882
Debug Support Protocol 857
Functions
GetMaximumProcessorIndex() 859
InvalidateInstructionCache() 874
RegisterExceptionCallback() 868
RegisterPeriodicCallback() 860
GUID 857
Interface Structure 857
Debugport device path 879
Debugport Protocol 875
Functions
Poll() 879
Read() 878
Reset() 876
Write() 877
GUID 875
Interface Structure 876
Decompress Protocol 897
Functions
Decompress() 899
GetInfo() 897
GUID 897
Interface Structure 897
Decompress() 899
decompression source code 2318
decompressor design 896
Defined GUID Partition Entry
Attributes 126
Partition Type GUIDs 125
Delete() 516

design overview 7

Desktop Management Interface (DMI), definition of 2406

Desktop Management Task Force (DMTF), definition of 2406

Device Handle, definition of 2406

Device Path
  for IDE disk 2190
  for legacy floppy 2189
  for secondary root PCI bus with PCI to PCI bridge 2191

Device Path Generation, Rules 331
  Hardware vs. Messaging Device Paths 333
  Housekeeping 331
  Media Device Path 333
  Other 334
    with ACPI _ADR 332
    with ACPI _HID and _UID 332

Device Path Instance, definition of 2406

Device Path Node, definition of 2406

Device Path Protocol 291
  GUID 292
    Interface Structure 292
device path protocol 291

Device Path, ACPI 296

Device Path, BIOS Boot Specification 334

Device Path, definition of 2407

Device Path, hardware
  memory-mapped 295
  PCCARD 294
  PCI 294
  vendor 295

Device Path, media 299, 325
  Boot Specification 330
  CD-ROM Media 326
  File Path Media 327
  hard drive 325
  Media Protocol 328
  Vendor-Defined Media 327

Device Path, messaging 299
  1394 302
  ATAPI 299
  FibreChannel 300
  I2O 306
  InfiniBand 308
IPv4 307
IPv6 307
MAC Address 306
SCSI 300
UART 308
UART flow control 310
USB 302
USB class 306
Vendor-Defined 309

Device Path, nodes
  ACPI Device Path 292
  BIOS Boot Specification Device Path 292
  End of Hardware Device Path 292
  End This Instance of a Device Path 293
  generic 293
  Hardware Device Path 292
  Media Device Path 292
  Messaging Device Path 292

Device Path, overview 291
device paths
  EFI simple pointer 464
  PS/2 mouse 465
  serial mouse 466
  USB mouse 467

DHCP packet 1047
Dhcp() 1054

DHCP4 Option Data
  Interface Structure 1464

DHCP4 Packet Data
  Interface Structure 1458, 1480

DisconnectController() 207
Discover() 1055

Disk I/O Protocol 543
  Functions
    ReadDisk() 336, 337, 338, 339, 357, 358, 359, 360, 535, 536, 537, 538, 539, 540, 544, 775, 776, 1224, 1225, 2082, 2083, 2090, 2091, 2092, 2093, 2094
    WriteDisk() 545, 1218, 1219, 1222, 1223, 1224, 1225, 1228
  GUID 334, 356, 359, 533, 543, 775, 1218, 1221, 1684, 2082, 2088
  Interface Structure 334, 356, 359, 534, 543, 775, 1218, 1221, 1684, 2082, 2088, 2132
  Revision Number 543

DIV 932
DIVU 933
document
attributes 4
audience 6
goals 4
purpose 1
driver binding protocol 361
driver diagnostics protocol 388
Driver Model Boot Services 177
Driver Signing 1689
DriverLoaded() 385
Dynamic Host Configuration Protocol (DHCP), definition of 2408

E
EBC Image, definition of 2408
EBC Instruction
  ADD 923
  AND 924
  ASHR 925
  BREAK 926
  CALL 928
  CMP 930
  CMPI 931
  DIV 932
  DIVU 933
  EXTNDB 934
  EXTNDD 935
  EXTNDW 936
  JMP 937
  JMP8 939
  LOADSP 939
  MOD 940
  MODU 941
  MOV 942
  MOVI 943
  MOVIn 944
  MOVn 945
  MOVREL 946
  MOVsn 947
  MUL 949
  MULU 950
  NEG 951
  NOT 952
  OR 953
  POP 954
POPn 955
PUSH 956
PUSHn 956
RET 957
SHL 958
SHR 959
STORESP 960
SUB 960
XOR 961

EBC instruction descriptions 11
EBC instruction encoding 922
EBC instruction operands 920
  direct operands 920
  immediate operands 921
  indirect operands 920
  indirect with index operands 920
EBC Instruction Set 923
EBC instruction set 923
EBC instruction syntax 921
EBC Interpreter Protocol 964
  Functions
    CreateThunk() 965
    GetVersion() 968
    RegisterICacheFlush() 967
    UnloadImage() 966
GUID 965
  Interface Structure 965
EBC Tools 969
EBC tools
  C coding convention 969
  debug support 973
  EBC C compiler 969
  EBC interface assembly instructions 969
  EBC linker 973
  EBC to EBC arguments calling convention 970
  EBC to native arguments calling convention 970
  function return values 970
  function returns 970
  image loader 973
  native to EBC arguments calling convention 970
  stack maintenance and argument passing 969
  thunking 971
  VM exception handling 974
EBC virtual machine 915
  architectural requirements 963
  runtime and software conventions 962
EFI Application 16, 17, 505
EFI Application, definition of 2423
EFI Boot Manager 506
EFI Boot Services Table 91
EFI Bus-Specific Driver Override Protocol
  functions
    GetDriver() 387
EFI Byte Code (EBC) 915
EFI Byte Code (EBC), definition of 2408
EFI Byte Code Virtual Machine 2
EFI Component Name Protocol 684
  functions
    GetControllerName() 394
    GetDriverName() 393
EFI Debug Support Protocol 856
EFI debug support table 881
EFI Debugport Protocol 875
EFI debugport variable 880
EFI DHCPv4 Protocol
  Functions
    Build() 1468
    GetModeData() 1455, 1476
    Parse() 1472, 1496
    Release() 1467, 1492, 1494
    RenewRebind() 1466, 1488, 1491
    Start() 1458, 1465, 1481, 1487
    Stop() 1468, 1495
    TransmitReceive() 1470
  GUID 1454, 1475
  Interface Structure 1454, 1475
EFI DHCPv4 Service Binding Protocol
  GUID 1453, 1474
EFI Directory Structure 506
EFI Driver 505
EFI Driver Binding Protocol
  functions
    Start() 368
    Stop() 377
    Supported() 363
EFI Driver Configuration Protocol
functions
  OptionsValid() 407
  SetOptions() 405
EFI Driver Diagnostics Protocol 684
 EFI Driver Diagnostics Protocol
  functions
    RunDiagnostics() 389
EFI Driver Model 1
 EFI driver model 8
 EFI Driver, definition of 2423
 EFI File, definition of 2408
 EFI Hard Disk, definition of 2408
 EFI Image 15, 505
 EFI Image handoff state 26
 IA-32 23
 EFI Image Header 15
 PE32+ image format 15
 EFI Image, definition of 2412
 EFI IPv4 Configuration Protocol
  Functions
    GetData() 1346
    Start() 1343
    Stop() 1345
 GUID 1343
 Interface Structure 1343
 EFI IPv4 Protocol
  Functions
    Cancel() 1341, 1378
    GetModeData() 1148, 1231, 1232, 1235, 1237, 1247, 1252, 1254, 1258, 1270, 1272, 1273,
      1278, 1325, 1358, 1427, 1434, 1435, 1436, 1439, 1440
    Groups() 1331, 1367
    Open() 1330, 1366
    Receive() 1339, 1377
    Route() 1332, 1368
    Transmit() 1333, 1371
 GUID 1324, 1357
 Interface Structure 1128, 1147, 1158, 1170, 1185, 1195, 1231, 1232, 1236, 1270, 1324, 1357,
   1427, 1438, 1498, 1512
 EFI IPv4 Service Binding Protocol
  GUID 1147, 1157, 1170, 1185, 1194, 1231, 1232, 1236, 1270, 1323, 1357, 1426, 1427, 1438,
   1497, 1498, 1511, 1512
 EFI MTFTP4 Protocol
  Functions
WriteFile() 1637

EFI MTFTPv4 Protocol

Functions
Configure() 1621, 1645
GetInfo() 1622
GetModeData() 1618
ParseOptions() 1631
ReadDirectory() 1639
ReadFile() 1632

Interface Structure 1617

EFI MTFTPv4 Service Binding Protocol
GUID 1617

EFI OS Loader 16, 505
EFI OS loader, definition of 2423
EFI partitioning scheme 120

EFI Platform Driver Override Protocol
functions
DriverLoaded() 385
GetDriver() 383
GetDriverPath() 384

EFI Runtime Services Table 91

EFI Scan Codes, SIMPLE_INPUT_INTERFACE 437

EFI Service Binding Protocol

Functions
CreateChild() 397
DestroyChild() 401

GUID 396

Interface Structure 397

EFI Specification 1
Design Overview 7
Goals 4
Target Audience 6

EFI System Table 91

EFI system table location 882

EFI Tables
EFI_BOOT_SERVICES 95
EFI_CONFIGURATION_TABLE 102
EFI_RUNTIME_SERVICES 100
EFI_SYSTEM_TABLE 94
EFI_TABLE_HEADER 92

EFI tables
EFI_IMAGE_ENTRY_POINT 91

EFI time 2178
EFI UDPv4 Protocol

Functions

- Cancel() 1300, 1600, 1615
- GetModeData() 1282, 1286, 1586, 1588, 1603, 1606
- Groups() 1590, 1607
- Poll() 1301, 1600, 1615
- Receive() 1297, 1598, 1614
- Route() 1288, 1591
- Transmit() 1292, 1293, 1593, 1608

GUID 1280, 1585, 1602

Interface Structure 1281, 1585, 1602

EFI USB Host Controller Protocol

functions

- AsyncInterruptTransfer() 790
- AsyncIsochronousTransfer() 797
- BulkTransfer() 788
- ClearRootHubPortFeature() 805
- ControlTransfer() 785
- GetRootHubPortNumber() 780
- GetRootHubPortStatus() 799
- GetState() 783
- IsochronousTransfer() 795
- Reset() 781
- SetRootHubPortFeature() 802
- SetState() 784
- SyncInterruptTransfer() 793

EFI, definition of 2408

EFI_ALLOCATE_TYPE 166

EFI_ARP_CONFIG_DATA 1445

EFI_ARP_FIND_DATA 1449

EFI_ARP_PROTOCOL 1443

EFI_ARP_SERVICE_BINDING_PROTOCOL 1442

EFI_AUTHENTICATION_INFO_PROTOCOL 1684

EFI_BIS_PROTOCOL 1082

EFI_BIS_SIGNATURE_INFO 1093

EFI_BIS_VERSION 1085

EFI_BLOCK_IO_MEDIA 554

EFI_BOOT_SERVICES table 95

EFI_BUS_SPECIFIC_DRIVER_OVERRIDE_PROTOCOL 386

EFI-compliant, definition of 2408

EFI_COMPONENT_NAME2_PROTOCOL 392

EFI_CONFIGURATION_TABLE 102

EFI_DECOMPRESS_PROTOCOL 897
EFI_DEVICE_PATH 292
EFI_DEVICE_PATH protocol 291
EFI_DEVICE_PATH_UTILITIES_PROTOCOL 334, 335, 336, 337, 338, 339, 340, 356, 357, 358, 359, 360
EFI_DHCP4_CALLBACK 1460, 1483, 1490
EFI_DHCP4_CONFIG_DATA 1459, 1477, 1482, 1486
EFI_DHCP4_EVENT 1462
EFI_DHCP4_HEADER 1463, 1481, 1484
EFI_DHCP4_LISTEN_POINT 1472
EFI_DHCP4_MODE_DATA 1455, 1477, 1478, 1480
EFI_DHCP4_PACKET 1458, 1480
EFI_DHCP4_PACKET_OPTION 1464
EFI_DHCP4_PROTOCOL 1453, 1454, 1455, 1456, 1457, 1458, 1460, 1461, 1465, 1466, 1467, 1468, 1469, 1470, 1471, 1472, 1473, 1474, 1475, 1481, 1483, 1487, 1488, 1489, 1490, 1491, 1495, 1496
EFI_DHCP4_SERVICE_BINDING_PROTOCOL 397, 1453, 1474
EFI_DHCP4_STATE 1456, 1479, 1485
EFI_DHCP4_TRANSMIT_RECEIVE_TOKEN 1471
EFI_DRIVER_BINDING_PROTOCOL 361
EFI_DRIVER_DIAGNOSTICS_PROTOCOL 388
EFI_DRIVER_DIAGNOSTIC_TYPE 391
EFI_EBC_PROTOCOL 964
EFI_EDID_ACTIVE_PROTOCOL 494
EFI_EDID_DISCOVERED_PROTOCOL 493
EFI_EVENT 149
EFI_FILE_INFO 530
  GUID 530
EFI_FILE_SYSTEM_INFO 531
  GUID 531
EFI_FILE_SYSTEM_VOLUME_LABEL 532
  GUID 532
EFI_GRAPHICS_OUTPUT_PROTOCOL_SET_MODE 490
EFI_GUID 181
EFI_HANDLE 181
EFI_HASH_PROTOCOL 2082, 2083, 2085, 2087, 2088, 2090, 2091, 2092, 2093, 2094, 2097
EFI_IMAGE_ENTRY_POINT 91, 222
EFI_INPUT_KEY 448
EFI_INTERFACE_TYPE 181
EFI_IP4_COMPLETION_TOKEN 1334, 1372, 1373, 1374, 1375
EFI_IP4_CONFIG_DATA 1327, 1361
EFI_IP4_CONFIG_PROTOCOL 1283, 1289, 1328, 1330, 1333, 1342, 1343, 1344, 1345, 1346, 1588, 1592, 1605
EFI_IP4_FRAGMENT_DATA 1337
EFI_IP4_HEADER 1336
| EFI_IP4_ICMP_TYPE             | 1329, 1364 |
| EFI_IP4_IPCONFIG_DATA         | 1347      |
| EFI_IP4_MODE_DATA             | 1326, 1359|
| EFI_IP4_OVERRIDE_DATA         | 1338      |
| EFI_IP4_PROTOCOL              | 1232, 1237, 1270, 1282, 1324, 1325, 1330, 1331, 1332, 1333, 1334, 1339, 1340, 1341, 1342, 1347, 1357, 1358, 1359, 1366, 1367, 1368, 1369, 1370, 1371, 1377, 1378, 1379, 1438, 1586 |
| EFI_IP4_RECEIVE_DATA          | 1335      |
| EFI_IP4_ROUTE_TABLE           | 1329, 1363|
| EFI_IP4_SERVICE_BINDING_PROTOCOL | 323, 396, 1147, 1157, 1170, 1185, 1194, 1230, 1232, 1236, 1270, 1323, 1324, 1356, 1357, 1426, 1438, 1497, 1498, 1510, 1511 |
| EFI_IP4_TRANSMIT_DATA         | 1337      |
| EFI_ISCSI_INITIATOR_NAME_PROTOCOL | 774      |
| EFI_LBA                       | 554       |
| EFI_LOADED_IMAGE Protocol     | 288       |
| EFI_LOCATE_SEARCH_TYPE        | 187       |
| EFI_MANAGED_NETWORK_COMPLETION_TOKEN | 1140 |
| EFI_MANAGED_NETWORK_CONFIG_DATA | 1134    |
| EFI_MANAGED_NETWORK_FRAGMENT_DATA | 1143 |
| EFI_MANAGED_NETWORK_PROTOCOL  | 1131      |
| EFI_MANAGED_NETWORK_RECEIVE_DATA | 1141   |
| EFI_MANAGED_NETWORK_SERVICE_BINDING_PROTOCOL | 1131 |
| EFI_MANAGED_NETWORK_TRANSMIT_DATA | 1142 |
| EFI_MEMORY_DESCRIPTOR        | 170       |
| EFIMEMORY_TYPE               | 166       |
| EFI_MTFTP4_ACK8_HEADER       | 1625, 1649|
| EFI_MTFTP4_ACK_HEADER        | 1625, 1649|
| EFI_MTFTP4_DATA8_HEADER      | 1625, 1649|
| EFI_MTFTP4_DATA_HEADER       | 1625, 1649|
| EFI_MTFTP4_ERROR_HEADER      | 1626, 1649|
| EFI_MTFTP4_OACK_HEADER       | 1625, 1649|
| EFI_MTFTP4_PACKET            | 1624, 1648|
| EFI_MTFTP4_PROTOCOL          | 1617, 1618, 1621, 1622, 1623, 1624, 1626, 1631, 1632, 1633, 1634, 1635, 1636, 1637, 1638, 1639, 1641, 1646, 1655, 1656, 1658, 1659, 1661, 1664 |
| EFI_MTFTP4_REQ_HEADER        | 1625, 1648|
| EFI_NETWORK_INTERFACE_TYPE   | 1041      |
| EFI_NETWORK_STATISTICS        | 1030      |
| EFI_OPEN_PROTOCOL_BY_CHILD_CONTROLLER | 194  |
| EFI_OPEN_PROTOCOL_BY_DRIVER   | 194, 198  |
| EFI_OPEN_PROTOCOL_BY_HANDLE_PROTOCOL | 194, 197 |
| EFI_OPEN_PROTOCOL_EXCLUSIVE   | 195, 198  |
| EFI_OPEN_PROTOCOL_GET_PROTOCOL | 194, 197 |
| EFI_OPEN_PROTOCOL_TEST_PROTOCOL | 194, 197 |
EFI_OPTIONAL_PTR 267
EFI_PARITY_TYPE 475
EFI_PCI_IO_PROTOCOL_ACCESS 696
EFI_PCI_IO_PROTOCOL_ATTRIBUTE_OPERATION 719
EFI_PCI_IO_PROTOCOL_CONFIG 696
EFI_PCI_IO_PROTOCOL_CONFIG_ACCESS 696
EFI_PCI_IO_PROTOCOL_IO_MEM 695
EFI_PCI_IO_PROTOCOL_POLL_IO_MEM 695
EFI_PCI_IO_PROTOCOL_WIDTH 695
EFI_PCI_ROOT_BRIDGE_IO_PROTOCOL_ACCESS 656
EFI_PCI_ROOT_BRIDGE_IO_PROTOCOL_IO_MEM 656
EFI_PCI_ROOT_BRIDGE_IO_PROTOCOL_POLL_IO_MEM 656
EFI_PCI_ROOT_BRIDGE_IO_PROTOCOL_WIDTH 655
EFI_PHYSICAL_ADDRESS 167
EFI_PLATFORM_DRIVER_OVERRIDE_PROTOCOL 381
EFI_PXE_BASE_CODE_CALLBACK_PROTOCOL 1080
EFI_PXE_BASE_CODE_CALLBACK_STATUS 1081
EFI_PXE_BASE_CODE_FUNCTION 1081
EFI_PXE_BASE_CODE_MODE 1043
EFI_PXE_BASE_CODE_MTFTP_INFO 1060
EFI_PXE_BASE_CODE_PROTOCOL 1041
EFI_PXE_BASE_CODE_TFTP_OPCODE 1060
EFI_RESET_TYPE 269
EFI_RUNTIME_SERVICES table 100
EFI_SERVICE_BINDING_PROTOCOL 396
EFI_SIMPLE_NETWORK_MODE 1019
EFI_SIMPLE_NETWORK_PROTOCOL 1017
EFI_SIMPLE_NETWORK_STATE 1021
EFI_SIMPLE_POINTER_MODE 461
EFI_SIMPLE_POINTER_STATE 463
EFI_STATUS codes ranges 2194
EFI_STATUS Error Codes 2194
EFI_STATUS Success Codes 2194
EFI_STATUS warning codes 2196
EFI_STOP_BITS_TYPE 475
EFI_SYSTEM_TABLE 94
EFI_TABLE_HEADER 92
EFI_TAPE_IO_PROTOCOL 533
EFI_TCP4_PROTOCOL 1280, 1281, 1282, 1286, 1288, 1289, 1290, 1292, 1293, 1297, 1298, 1299, 1300, 1301
EFI_TCP4_SERVICE_BINDING_PROTOCOL 396, 1280
EFI_TIME 258
EFI_TIME_CAPABILITIES 260
EFI_UDP4_COMPLETION_TOKEN 1290, 1292, 1299, 1593, 1609
EFI_UDP4_CONFIG_DATA 1285, 1286, 1587
EFI_UDP4_FRAGMENT_DATA 1296, 1596, 1611
EFI_UDP4_PROTOCOL 1584, 1585, 1586, 1588, 1589, 1590, 1591, 1593, 1596, 1598, 1599, 1600, 1601, 1602
EFI_UDP4_RECEIVE_DATA 1295, 1595, 1610
EFI_UDP4_SERVICE_BINDING_PROTOCOL 396, 1584
EFI_UDP4_SESSION_DATA 1596, 1611
EFI_UDP4_TRANSMIT_DATA 1296, 1597
EFI_UNICODE_COLLATION_PROTOCOL 904
EFI_USB2_HC_PROTOCOL 778
EFI_USB_IO Protocol 809
EFI_VIRTUAL_ADDRESS 172
El Torito 504, 507, 508
EnableCursor() 460
End of Hardware Device Path, definition of 2408
Enhanced Mode (EM), definition of 2408
error codes 2194
Event Services 144
   function list 144
   functions
      CheckEvent() 159
      CloseEvent() 156
      CreateEvent() 148
      SignalEvent() 157
      WaitForEvent() 158
   overview 144
event, definition of 2409
Exit() 223
ExitBootServices() 225
Extensible Firmware Interface Specification 1
EXTNDB 934
EXTNDD 935
EXTNDW 936

F
FAT file system 504
FAT File System, definition of 2409
FAT variants 505
FatToStr() 908
File Allocation Table (FAT), definition of 2409
file attribute bits, EFI_FILE_INFO 530
File Attributes, EFI_FILE_PROTOCOL 515
File Handle Protocol 512

Functions
- Close() 516
- Delete() 516
- EFI_FILE_SYSTEM_INFO 531, 532
- EFI_GENERIC_FILE_INFO 530
- Flush() 529
- GetInfo() 527
- GetPosition() 526
- Open() 514
- Read() 517
- SetInfo() 528
- SetPosition() 525
- Write() 518

Interface Structure 513
Revision Number 512

file names 505
file system format 504, 505
File System Protocol 510
Fill Header 2272
Firmware Interrupts level 145
firmware menu 14
Firmware, definition of 2409
Flush() 529, 674, 716
FlushBlocks() 559, 565
Free() 1088
FreeBuffer() 674, 715
FreePages() 168
FreePool() 174

G
Geometric Shapes Code Chart 453
Get Config Info 2247
Get Init Info 2244
Get State 2234
Get Status 2270
GetAttributes() 675
GetBarAttributes() 720
GetBootObjectAuthorizationCertificate() 1089
GetBootObjectAuthorizationCheckFlag() 1090
GetBootObjectAuthorizationUpdateToken() 1091
GetControl() 480
  control bits 481
GetControllerName() 394
GetDriver() 383, 387
GetDriverName() 393
GetDriverPath() 384
GetInfo() 527, 897
GetLocation() 717
GetMaximumProcessorIndex() 859
GetMemoryMap() 169
GetNextHighMonotonicCount() 269
GetNextMonotonicCount() 230
GetNextVariableName() 239
GetPosition() 526
GetRootHubPortNumber() 780
GetRootHubPortStatus() 799
  PortChangeStatus bit definition 800
  PortStatus bit definition 800
GetSignatureInfo() 1092
GetState() 463, 783
GetStatus() 1034
GetTime() 257
GetVariable() 236
GetVersion() 968
GetWakeupTime() 262
globally unique identifier, definition of 2410
glossary 2403
GPT See GUID Partition Table
GUID Partition Entry 124
GUID Partition Entry, definition of 2410
GUID Partition Table 507
  GPT 120, 121, 123, 124, 125, 504, 507, 508
GUID Partition Table Header 123, 508
GUID Partition Table Header, definition of 2410
GUID Partition Table, definition of 2410
GUID Partition, definition of 2411
GUID, definition of 2410

H
Handle, definition of 2411
HandleProtocol() 188
Hardware Device Path, definition of 2411
Hash
  Hash 2081
Headless system 291
_HID 296
_HID, definition of 2403
Huffman code generation 894
Huffman coding 2288
HYPERLINK "ch10.doc" "SIMPLE_TEXT_OUTPUT"

image: implementation control sequences 2183

I
IA-32
EFI Image handoff state 23
ICMP error packet 1047
ICMP Message Types and Codes
Data Structure 1329, 1364
IDE disk device path 2191
Image Handle, definition of 2411
Image Handoff State, definition of 2411
Image Header, definition of 2411
Image Services
function list 217
functions
EFI_IMAGE_ENTRY_POINT 222
Exit() 223
ExitBootServices() 225
LoadImage() 217
StartImage() 220
UnloadImage() 221
overview 216
Image, definition of 2412
images
loading 14
implementation requirements
general 62
required elements 62
Initialize 2250
Initialize() 1023, 1084
InstallConfigurationTable() 231
InstallMultipleProtocolInterfaces() 214
InstallProtocolInterface() 179
instruction summary
  EFI byte code virtual machine 2334
Intel Architecture-32 (IA-32), definition of 2412
Intel® Itanium™ Architecture, definition of 2412
interfaces
  general categories 18
  purpose 17
Interpreter, definition of 2412
Interrupt Enables 2256
InterruptStatus interrupt bit mask settings 1035
InvalidateInstructionCache() 874
Io.Read() 665, 706
Io.Write() 665, 706
IP filter operation 1066
IP4 Protocol
  Functions
    Poll() 1341, 1379
IPv4 Fragment Data
  Data Structure 1337
IPv4 Header
  Data Structure 1336
IPv4 IOCTL Data
  Data Structure 1327, 1361
IPv4 Mode Data
  Data Structure 1326, 1359
IPv4 Override Data
  Data Structure 1338
IPv4 Receive Data
  Data Structure 1335
IPv4 Route Table
  Data Structure 1329, 1363
IPv4 Transmit Data
  Data Structure 1337
ISO-9660 508
IsochronousTransfer() 795
Itanium architecture
  EFI Image handoff state 26
  firmware specifications 2402
  platforms 2402
  requirements, related to this specification 2402
J
JMP 937
JMP8 939

L
LAN On Motherboard (LOM), definition of 2412
LBA See Logical Block Address
legacy floppy device path 2190
legacy interfaces 5
legacy Master Boot Record 115
and GPT Partitions 118
Partition Record 116
legacy MBR 504
legacy OS 6
Legacy Platform, definition of 2413
legacy systems, support of 9
Little Endian, definition of 2413
Load File Protocol 1051
  Functions
    LoadFile() 500
  GUID 500
  Interface Structure 500
Loaded Image Protocol 288
  functions
    Unload() 289
  GUID 288
  Interface Structure 288
  Revision Number 288
Loaded Image, definition of 2413
LoadFile() 500
LoadImage() 217
LOADSP 939
LocateDevicePath() 190
LocateHandle() 186
LocateHandleBuffer() 210
LocateProtocol() 213
long file names 505
Long File Names (LFN), definition of 2413
LZ77 coding 2288

M
Machine Check Abort (MCA), definition of 2413
Managed Network Protocol

Functions
- Cancel() 1145
- Configure() 1135
- GetModeData() 1133
- Groups() 1138
- McastIpToMac() 1137
- Poll() 1146
- Receive() 1144
- Transmit() 1139

GUID 1132
Interface Structure 1132

Managed Network Service Binding Protocol

GUID 1131
Map() 670, 711
Master Boot Record 504
Master Boot Record (MBR), definition of 2413
MAX_MCAST_FILTER_CNT 1021
MBR See Master Boot Record
MCast IP To MAC 2266
MCastIPtoMAC() 1032
Media Device Path, definition of 2414
media formats 509
Mem.Read() 664, 704
Mem.Write() 664, 704

Memory Allocation Services
function list 163
functions
- AllocatePages() 165
- AllocatePool() 173
- FreePages() 168
- FreePool() 174
- GetMemoryMap() 169

overview 163
Memory Attribute Definitions 171
memory map 163
Memory Map, definition of 2414
Memory Type, definition of 2414
memory type, usage

after HYPERLINK l "ExitBootServices"
08d0c9ea79f9bace118c8200aa004ba90b02000000800000011000000045007800690
0740042006f006f00740005300650072007600690063006500730000000ExitBootServices() 163
Messaging Device Path, definition of 2414
MetaIMatch() 906
migration requirements 9
  EFI support on a legacy platform 10
  legacy OS support 9
migration, from legacy systems 9
Miscellaneous Boot Services
  overview 226
Miscellaneous Runtime Services
  overview 267
Miscellaneous Services
  function list 226, 268
  functions
    CalculateCrc32() 232
    CopyMem() 228
    GetNextHighMonotonicCount() 269
    GetNextMonotonicCount() 230
    InstallConfigurationTable() 231
    ResetSystem() 247, 268, 271, 279
    SetMem() 229
    SetWatchdogTimer() 227
    Stall() 228
MOD 940
MODU 941
MOV 942
MOVI 943
MOVIn 944
MOVn 945
MOVREL 946
MOVsn 947
Mtftp() 1058
MTFTP4 Packet Definitions 1624, 1648
MUL 949
Multicast Trivial File Transfer Protocol (MTFTP), definition of 2415
MULU 950

N
Name Space
  EFI device path 2193
Name space 291
Name Space, definition of 2415
Native Code, definition of 2415
natural indexing 918
NEG 951
Network Boot Program, definition of 2415
Network Bootstrap Program (NBP), definition of 2415
Network Interface Card (NIC), definition of 2415
Network Interface Identifier Protocol 1039
  GUID 1039
  Interface Structure 1039
  Revision Number 1039
nonvolatile storage 737
NOT 952
NvData 2267
NvData() 1033
NVRAM variables 70

O
opcode summary
  EFI byte code virtual machine 2334
Open Modes, EFI_FILE_PROTOCOL 515
Open() 514
OpenProtocol() 191
OpenProtocolInformation() 201
OpenVolume() 511
operating system loader, definition of 2423
Option ROM 5
option ROM 9, 915
  EBC 916
  legacy 916
  relocatable image 916
  size restrictions 916
option ROM formats 975
OptionsValid() 407
OR 953
OS loader, definition of 2423
OS Loader, EFI 16
OS network stacks 2201
OutputString() 451
overview of design 7
P
Page Memory, definition of 2416
partition discovery 507
Partition Discovery, definition of 2416
partitioning scheme, EFI 120
PCANSI terminals, and SIMPLE_TEXT_OUTPUT 2183
PCI bus driver responsibilities 729
PCI Bus Driver, definition of 2416
PCI bus drivers 685
PCI Bus, definition of 2416
PCI Configuration Space, definition of 2417
PCI Controller, definition of 2417
PCI device driver responsibilities 730
PCI Device Driver, definition of 2417
PCI device drivers 690
PCI device paths 724
PCI Device, definition of 2417
PCI driver initialization 683
PCI driver model 683
PCI Enumeration, definition of 2417
PCI Function, definition of 2417
PCI Host Bus Controller, definition of 2417
PCI hot-plug events 737
PCI I/O Protocol 692
Functions
  AllocateBuffer() 713
  Attributes() 718
  CopyMem() 709
  Flush() 716
  FreeBuffer() 715
  GetBarAttributes() 720
  GetLocation() 717
  Io.Read() 706
  Io.Write() 706
  Map() 711
  Mem.Read() 704
  Mem.Write() 704
  Pci.Read() 708
  Pci.Write() 708
  PollIo() 703
  PollMem() 701
  SetBarAttributes() 723
  Unmap() 713
GUID 693
  Interface Structure 693
PCI Option ROM, definition of 2418
PCI option ROMs 726
PCI root bridge device paths 680
PCI Root Bridge I/O Protocol 654
  Functions
    AllocateBuffer() 672
    Configuration() 678
    CopyMem() 668
    Flush() 674
    FreeBuffer() 674
    GetAttributes() 675
    Io.Read() 665
    Io.Write() 665
    Map() 670
    Mem.Read() 664
    Mem.Write() 664
    Pci.Read() 666
    Pci.Write() 666
    PollIo() 662
    PollMem() 661
    SetAttributes() 676
    Unmap() 671
  GUID 413, 654
  Interface Structure 654
PCI root bridge I/O support 649
PCI Root Bridge, definition of 2418
PCI Segment, definition of 2418
Pci.Read() 666, 708
Pci.Write() 666, 708
PE32+ image format 15
platform driver override protocol 381
plug and play option ROMs
  and boot services 17
pointer movement 2284
Poll() 879
PollIo() 662, 703
PollMem() 661, 701
Pool Memory, definition of 2418
POP 954
POKn 955
Preboot Execution Environment (PXE), definition of 2418
prerequisite specifications 2402

Protocol

11.7 Graphics Output Protocol 298
23.4 PXE Base Code Callback 1045, 1053, 1079
ARP 3, 396, 398, 399, 400, 401, 402, 403, 1042, 1043, 1044, 1046, 1049, 1050, 1068, 1069, 1070, 1081, 1288, 1333, 1442, 1443, 1444, 1445, 1446, 1447, 1448, 1449, 1450, 1451, 1452, 1453, 1592
ARP Service Binding 1442
Block I/O 552, 560
Boot Integrity Services 1082
Boot Integrity Services (BIS) 1044, 1082
Console I/O 183, 436
Debug Support 857
Debugport 875
Decompress 897
Device Path 291
Disk I/O 543
EBC Interpreter 964
EFI DHCPv4 Service Binding 1453, 1457, 1459, 1474
EFI IPv4 3, 1147, 1157, 1170, 1230, 1232, 1269, 1283, 1323, 1324, 1325, 1327, 1328, 1329, 1330, 1331, 1332, 1333, 1334, 1335, 1336, 1340, 1342, 1343, 1344, 1345, 1346, 1347, 1356, 1357, 1358, 1359, 1360, 1361, 1365, 1366, 1367, 1368, 1369, 1371, 1372, 1373, 1377, 1379, 1438, 1453, 1454, 1455, 1584, 1585, 1586, 1587, 1588, 1605, 1618
EFI MTFTPv4 3, 1415, 1417, 1418, 1420, 1423, 1424, 1425, 1426, 1534, 1536, 1545, 1616, 1617, 1618, 1619, 1621, 1622, 1626, 1631, 1633, 1634, 1635, 1636, 1637, 1638, 1639, 1640, 1641, 1645, 1654, 1656, 1658, 1659, 1660, 1661, 1662, 1664, 1665
EFI MTFTPv4 Service Binding 1616
EFI Service Binding 396, 1131, 1442
EFI TCPv4 3, 1280, 1281, 1283, 1286, 1287, 1289, 1290, 1291, 1293, 1297, 1298, 1300
EFI TCPv4 Service Binding 1280
EFI UD Pv4 3, 1584, 1585, 1586, 1587, 1588, 1589, 1590, 1591, 1592, 1593, 1594, 1595, 1596, 1597, 1598, 1599, 1616, 1618, 1641
File Handle 512
File System 510
Load File 1051
Loaded Image 288
Managed Network 3, 1131, 1444
Managed Network Service Binding 3, 1131
Network Interface Identifier 1038, 1039, 1040, 1050, 1051
PCI I/O 692
PCI Root Bridge I/O 654
PXE Base Code 1041
PXE Base Code Callback 1080
Serial I/O 472
Simple File System 510
Simple Input 436, 446
Simple Network 1017, 1029, 1032, 1033, 1038, 1039, 1041, 1051, 1132
Simple Pointer 461
Unicode Collation 904
Protocol Handler Services
function list 175
functions 175
  CloseProtocol() 199
  ConnectController() 202
  DisconnectController() 207
  HandleProtocol() 188
  InstallMultipleProtocolInterfaces() 214
  InstallProtocolInterface() 179
  LocateDevicePath () 190
  LocateHandle() 186
  LocateHandleBuffer() 210
  LocateProtocol() 213
  OpenProtocol() 191
  OpenProtocolInformation() 201
  ProtocolsPerHandle() 209
  RegisterProtocolNotify() 185
  ReinstallProtocolInterface() 183
  UninstallMultipleProtocolInterfaces() 215
  UninstallProtocolInterface() 182
overview 175
Protocol Handler, definition of 2419
Protocol Interface, definition of 2419
Protocol Revision Number, definition of 2419
Protocol, definition of 2419
protocols 45
  code illustrating 46
  construction of 46
  EFI_BUS_SPECIFIC_DRIVER_OVERRIDE_PROTOCOL 386
  EFI_DEVICE_PATH 291
  EFI_DRIVER_BINDING_PROTOCOL 361
  EFI_DRIVER_DIAGNOSTICS_PROTOCOL 388
  EFI_PLATFORM_DRIVER_OVERRIDE_PROTOCOL 381
  EFI_USB2_HC_PROTOCOL 778
  EFI_USB_IO Protocol 809
list of 47
UGA protocols 483
ProtocolsPerHandle() 209
PUSH 956
PUSHn 956
PXE Base Code Callback Protocol 1080
  Functions
    Callback() 1080
    GUID 1080
    Interface Structure 1080
    Revision Number 1080
PXE Base Code Protocol 1041
  Functions
    Arp() 1068
    Dhcp() 1054
    Discover() 1055
    Mtftp() 1058
    SetIpFilter() 1067
    SetPackets() 1072
    SetParameters() 1070
    SetStationIp() 1071
    Start() 1051
    Stop() 1053
    UdpRead() 1064
    UdpWrite() 1062
    GUID 1041
    Interface Structure 1042
    Revision Number 1041
PXE boot server bootstrap types 1056
PXE tag definitions for EFI 1050

Q
QueryCapsuleCapsule() 279
QueryMode() 455

R
Read() 517, 878
Read(), SERIAL_IO 482
ReadBlocks() 556, 561
ReadDisk() 544
ReadKeyStroke() 447
Read-Only Memory (ROM), definition of 2420
Receive 2279
Receive Filters 2258
Receive() 1037
ReceiveFilters() 1025
ReceiveFilterSetting bit mask values 1021
RegisterExceptionCallback() 868
RegisterICacheFlush() 967
RegisterPeriodicCallback() 860
RegisterProtocolNotify() 185
ReinstallProtocolInterface() 183
Reset, PXE 2253
Reset, UNDI 2253
Reset(), Debugport Protocol 876
Reset(), EFI_BLOCK_IO 556, 560
Reset(), EFI_SIMPLE_POINTER 462
Reset(), SERIAL_IO 477
Reset(), Simple Network Protocol 1024
Reset(), SIMPLE_INPUT 446
Reset(), SIMPLE_TEXT_OUTPUT 451
Reset(), USB Host Controller 781
ResetSystem() 247, 268, 271, 279
RestoreTPL() 162
RET 957
RunDiagnostics() 389
Runtime Services 143, 233
  Miscellaneous Runtime Services 267
  Time Services 257
  Variable Services 236
  Virtual Memory Services 264
runtime services 7, 18
Runtime Services Driver, definition of 2423
Runtime Services Table, definition of 2420
Runtime Services Table, EFI 91
Runtime Services, definition of 2420

S
SAL, definition of 2420
SAS Boot 311, 313
SCSI Pass Thru device paths 754
SCSI Pass Thru Protocol
  using 2285
Secondary Root PCI Bus with PCI to PCI Bridge Device Path 2192
Security
  Driver Signing 1689
    Hash 2081, 2082, 2083, 2084, 2085, 2088, 2091, 2092, 2093, 2094, 2095, 2097
Serial I/O Protocol 472
Functions
  GetControl() 480
  Read() 482
  Reset() 477
  SetAttributes() 478
  SetControl() 479
  Write() 481
GUID 472
Interface Structure 473
Revision Number 472
SERIAL_IO_MODE 474
services 17
SetAttribute() 457
SetAttributes() 478, 676
SetBarAttributes() 723
SetControl() 479
  control bits 480
SetCursorPosition() 459
SetInfo() 528
SetIpFilter() 1067
SetMem() 229
SetMode() 456, 490, 495
SetOptions() 405
SetPackets() 1072
SetParameters() 1070
setPosition() 525
SetRootHubPortFeature() 802
setState() 784
SetStationIp() 1071
SetTime() 261
SetTimer() 160
SetVariable() 241
SetVirtualAddressMap() 265
SetWakeupTime() 263
SetWatchdogTimer() 227
SHL 958
SHR 959
Shutdown 2255
Shutdown() 1025, 1087
SignalEvent() 157
Simple File System Protocol 510
  functions
    OpenVolume() 511
GUID 510
Interface Structure 510
Revision Number 510
Simple Input Protocol 436, 446
Functions
  ReadKeyStroke() 447
  Reset() 446
GUID 446
Interface Structure 446
Scan Codes for 437
Simple Network Protocol 1017, 1041, 1051
Functions
  GetStatus() 1034
  Initialize() 1023
  MCastIPtoMAC() 1032
  NVData() 1033
  Receive() 1037
  ReceiveFilters() 1025
  Reset() 1024
  Shutdown() 1025
  Start() 1021
  StationAddress() 1028
  Statistics() 1029
  Stop() 1022
  Transmit() 1035
GUID 1017
Interface Structure 1017
Revision Number 1017
Simple Pointer Protocol 461, 2284
Functions
  GetState() 463
  Reset() 462
GUID 461
Protocol Interface Structure 461
Simple Text Output Protocol
Functions
  ClearScreen() 458
  EnableCursor() 460
  OutputString() 451
  Querymode() 455
  Reset() 451
  SetAttribute() 457
  SetCursorPosition() 459
Setmode() 456
TestMethod() 454
GUID 449
Interface Structure 449
SIMPLE_INPUT protocol, implementation 2180
SIMPLE_TEXT_OUTPUT protocol, implementation 2180
SIMPLE_TEXT_OUTPUT_MODE 450
SMBIOS, definition of 2421
specifications, other 2402
specifications, prerequisite 2402
Stall() 228
StandardError 448
StandardError, definition of 2422
Start 2236
Start() 368, 1021
Start(), PXE Base Code Protocol 1051
StartImage() 220
Station Address 2261
StationAddress() 1028
Statistics 2263
Statistics() 1029
Status Codes, definition of 2422
Stop 2243
Stop() 377, 1022
Stop(), PXE Base Code Protocol 1053
STORESP 960
StriColl() 905
StrLwr() 907
StrToFat() 909
StrUpf() 907
SUB 960
success codes 2194
Supported() 363
SyncInterruptTransfer() 793
System Abstraction Layer (SAL), definition of 2422
System Management BIOS (SMBIOS), definition of 2422
System Partition 504, 506
system partition 7
System Table, definition of 2422
System Table, EFI 91

T
table-based interfaces 7
Task Priority Level (TPL), definition of 2422

task priority levels
  general 144
  restrictions 145
  usage 144

Task Priority Services 144
  function list 144
  functions
    RestoreTPL() 162
    overview 144

terminology, definitions 2403

TestString() 454

TFTP error packet 1047

Time Format, definition of 2422

Time Services
  function list 257
  functions
    GetTime() 257
    GetWakeupTime() 262
    SetTime() 261
    SetWakeupTime() 263
    overview 257

Timer Services 144
  function list 144
  functions
    SetTimer() 160
    overview 144

TPL restrictions 145

TPL See task priority levels

TPL_APPLICATION level 144, 145

TPL_HIGH_LEVEL 145

TPL_NOTIFY level 145

Transmit 2276

Trivial() 1035

Trivial File Transport Protocol (TFTP), definition of 2423

U

UDP port filter operation 1066

UDP4 Service Binding Protocol
  GUID 1280, 1584, 1602

UdpRead() 1064

UdpWrite() 1062

UGA Draw Protocol
Functions
  Blt() 491
  SetMode() 490, 495
GUID 485
  protocol interface structure 485
UGA protocols 483
  _UID 296
  _UID, definition of 2403
UNDI C definitions 2209
UNDI CDB 2207
UNDI CDB field definitions 2207
UNDI command descriptor block 2207
UNDI command format 2206
UNDI commands 2231
  Fill Header 2272
  Get Config Info 2247
  Get Init Info 2244
  Get State 2234
  Get Status 2270
  Initialize 2250
  Interrupt Enables 2256
  issuing 2206
  linking & queuing 2233
  MCast IP To MAC 2266
  NvData 2267
  Receive 2279
  Receive Filters 2258
  Reset 2253
  Shutdown 2255
  Start 2236
  Station Address 2261
  Statistics 2263
  Stop 2243
  Transmit 2276
UNDI Specification
  Definitions 2197
  driver types 2201
  Referenced Specifications 2198
UNDI Specification, 32/64-Bit 2197
Unicode Collation Protocol 904
Functions
  FatToStr() 908
  MetaiMatch() 906
StriColl() 905
StrLwr() 907
StrToFat() 909
StrUpr() 907
GUID 904
Interface Structure 904
Unicode control characters, supported 436
UNICODE DRAWING CHARACTERS 452
Unicode, definition of 2424
UninstallMultipleProtocolInterfaces() 215
UninstallProtocolInterface() 182
Universal Graphics Adapter protocols 483
Universal Network Device Interface (UNDI), definition of 2424
Universal Serial Bus (USB), definition of 2424
Unload() 289
UnloadImage() 221, 966
Unmap() 671, 713
Update Capsule 271
UpdateBootObjectAuthorization() 1096
   Manifest Syntax 1097
UpdateCapsule() 271
USB Bus Driver 807
   Bus Enumeration 808
   Driver Binding Protocol 807
   Entry Point 807
   Hot-Plug Event 807
USB Bus Driver, definition of 2424
USB Bus, definition of 2424
USB Controller, definition of 2424
USB Device Driver 808
   Driver Binding Protocol 808
   Entry Point 808
USB Device Driver, definition of 2425
USB Device, definition of 2425
USB Driver Model 806
USB Enumeration, definition of 2425
USB Host Controller Protocol 778
   GUID 778
   Interface Structure 778
USB host controller protocol 778
USB Host Controller, definition of 2425
USB hub port change status bitmap 801
USB hub port status bitmap 800
USB Hub, definition of 2425
USB I/O Protocol functions
  UsbAsyncInterruptTransfer () 814
  UsbAsyncIsochronousTransfer () 820
  UsbBulkTransfer () 813
  UsbControlTransfer() 811
  UsbGetConfigDescriptor () 823
  UsbGetDeviceDescriptor () 822
  UsbGetEndpointDescriptor() 826
  UsbGetInterfaceDescriptor () 825
  UsbGetStringDescriptor() 828
  UsbGetSupportedLanguages() 829
  UsbIsochronousTransfer () 819
  UsbPortReset() 830
  UsbSyncInterruptTransfer () 818
USB I/O protocol 809
  GUID 809
    Interface Structure 809
USB Interface, definition of 2425
USB port feature 803
USB transfer result error codes 812
UsbAsyncInterruptTransfer() 814
UsbAsyncIsochronousTransfer () 820
UsbBulkTransfer() 813
UsbControlTransfer() 811
UsbGetConfigDescriptor() 823
UsbGetDeviceDescriptor () 822
UsbGetEndpointDescriptor() 826
UsbGetInterfaceDescriptor () 825
UsbGetStringDescriptor() 828
UsbGetSupportedLanguages() 829
UsbIsochronousTransfer () 819
UsbPortReset() 830
UsbSyncInterruptTransfer () 818

V
Variable Attributes 237
Variable Services
  function list 236
  functions
    GetNextVariableName() 239
    GetVariable() 236
SetVariable() 241
overview 236
variables
global 81
non-volatile 82
VerifyBootObject() 1104
Manifest Syntax 1105
VerifyObjectWithCredential() 1110
Manifest Syntax 1111
virtual machine 915
registers 917
Virtual Memory Services
function list 264
functions
ConvertPointer() 266
SetVirtualAddressMap() 265
overview 264
VM, definition of 2425

W
WaitForEvent() 158
warning codes 2196
Watchdog timer, definition of 2426
WIN_CERTIFICATE 1693, 1694, 1695, 1696
Wired for Management (WfM), definition of 2426
Write() 518, 877
Write(), SERIAL_IO 481
WriteBlocks() 558, 563
WriteDisk() 545

X
x64 27
XOR 961