Embracing Modularity and Boot-Time Configuration

Faster TTM with Tiano-based Solutions

UEFI Fall 2023 Developers Conference & Plugfest
October 9-12, 2023
Andrei Warkentin (Intel)
Agenda

• Background
• FdtBusDxe
Background

• UEFI has a flexible driver model and a rich set of bus and I/O interfaces.

• No good mechanism to implement platform (non-discoverable) devices as part of bring-up – fiddly, hand-rolled, etc. Other implementations do better.

• Tiano is the reference UEFI implementation. Tiano == UEFI, yet UEFI != Tiano. Tiano gaps are frequently claimed as spec gaps... Time to fix the gaps!
Problem #1

• Compile-time definitions for platform devices.
  – DSC/FDF choose platform-specific drivers.
  – #defines, Pcds guide driver behavior

• Hand-rolled platform description mechanisms
  – https://github.com/tianocore/edk2-platforms/blob/master/Silicon/Marvell/Armada7k8k/Library/Armada7k8kSoCDescLib/Armada7k8kSoCDescLib.c
Problem #2

Not clearly following driver model.

- Some drivers just install protocols
  - [https://github.com/tianocore/edk2-platforms/blob/master/Silicon/Broadcom/Bcm283x/Drivers/Bcm2838RngDxe/Bcm2838RngDxe.c](https://github.com/tianocore/edk2-platforms/blob/master/Silicon/Broadcom/Bcm283x/Drivers/Bcm2838RngDxe/Bcm2838RngDxe.c)

- Some drivers are libraries consumed by a generic driver
  - PciHostBridgeLib consumed by PciHostBridgeDxe
  - SerialPortLib consumed by SerialDxe

- Some drivers publish a driver binding protocol, then install the one handle this protocol supports.
  - [https://github.com/tianocore/edk2-platforms/blob/master/Platform/RaspberryPi/Drivers/DwUsbHostDxe/DriverBinding.c](https://github.com/tianocore/edk2-platforms/blob/master/Platform/RaspberryPi/Drivers/DwUsbHostDxe/DriverBinding.c)

- A few common IP blocks can use NonDiscoverablePciDeviceDxe to bind existing PCIe drivers.
Problem #3

Monolithic drivers are hard.

- Some devices aren’t simple. They are a hierarchy and may have complex dependency outside of the immediate IP blocks.
- A NIC is a combination of:
  - Board-specific PHY.
  - Possibly GPIO/I2C and power resources.
  - NIC engine, possibly with per-port/instance resources.
- A video framebuffer could be a combination of:
  - An ISP with multiple video inputs (e.g. pixel format conversion)
  - An LCD controller.
  - An HDMI encoder (on I2C)
  - EDID source (on another I2C)
  - GPIOs, power rails, clocks.
  - Multiple video outputs/sinks.
- No code reuse.
- Hard to tweak – everything is backed in and platform wiring specific.
Problem #4

• Minor SoC variants require custom firmware image builds.
  – E.g. SoC 7xxx has half of the SoC8xxx I/Os...

• Minor board variants require custom firmware image builds (different PHY, different PCIe segment configs)
U-Boot Suggests a Solution

- Most drivers in U-Boot follow the U-Boot device driver model.
- On platforms with a flattened device tree, the latter can be used for driver binding and configuration.
What is Device Tree?

- A marshalling format for a hierarchical key-value store.
- [https://www.devicetree.org/](https://www.devicetree.org/)
- Came from PowerPC, embraced by Arm and RISC-V platforms.
- Typically used to describe hardware to an OS in vertically-integrated (“embedded”) environments, where it is used in place of ACPI.
Is This DT vs ACPI Again?

• This is not about passing DT to an OS.
  – This is a choice dictated by product segment, how crazy/broken/advanced your hardware is, etc.

• Incidentally, many things that make DT not particularly great for general purpose OS consumption make it great for describing hardware to firmware.
  – No abstraction, no byte code – raw data.
  – Can describe whatever the driver developers need.
  – Think of it as a PI HOB on steroids.
An Example

genet: ethernet@7d580000 {
    compatible = "brcm,bcm2711-genet-v5";
    reg = <0x0 0x7d580000 0x100000>;
    #address-cells = <0x1>;
    #size-cells = <0x1>;
    interrupts = <GIC_SPI 157 IRQ_TYPE_LEVEL_HIGH>,
                 <GIC_SPI 158 IRQ_TYPE_LEVEL_HIGH>;
    status = "disabled";

    phy-handle = <&phy1>;
    phy-mode = "rgmii-rxid";
    status = "okay";

    genet_mdio: mdio@e14 {
        compatible = "brcm,genet-mdio-v5";
        reg = <0xe14 0x8>;
        reg-names = "mdio";
        #address-cells = <0x1>;
        #size-cells = <0x0>;

        phy1: ethernet-phy@1 {
            /* NO PHY interrupt */
            reg = <0x1>;
        };
    };
};
How is DT Used in Tiano Today?

• Patched and passed to OS loaders when booting without ACPI (AArch64, Arm, RISCV64)
• Manually traversed in drivers.
• EmbeddedPkg/Drivers/FdtClientDxe

```c
STATIC FDT_CLIENT_PROTOCOL mFdtClientProtocol = {
    GetNodeProperty,
    SetNodeProperty,
    FindCompatibleNode,
    FindNextCompatibleNode,
    FindCompatibleNodeProperty,
    FindCompatibleNodeReg,
    FindMemoryNodeReg,
    FindNextMemoryNodeReg,
    GetOrInsertChosenNode,
};
```
A Few More Points

• DT blobs are almost always used by prior state firmware on Arm (TF-A) and RISC-V (OpenSBI) and passed to Tiano Sec/PrePi.

• Almost all platforms start with DT and then (possibly) add ACPI support, so the DT always exists in some form, at least to support early Linux enablement.

• ...but DT is not arch-specific. It can be trivially used on x86 to replace any other bespoke mechanism to describe hardware to Tiano platform drivers!
Introducing FdtBusDxe
FdtBusDxe

• A UEFI bus driver (e.g. similar to PciBus)
• Started at Intel, in the process of being open sourced (BSD).
• RISE Project under the firmware Working Group.
  – https://wiki.riseproject.dev/
• Goal is to upstream to edk2 once the design settles down and there is sufficient review.
FdtBusDxe

- Binds to DT top level (root) handle
- Binds to simple-bus nodes (like ACPI0004 containers)
- Exposes EFI_DT_IO_PROTOCOL
  - Common cached properties (Name, Model, Status)
  - Properties lookup, child device processing
  - Device register access (Poll, Read, Write, Copy)
  - DMA operations (Map, Unmap, Allocate, Free)
- FdtClientDxe replacement
  - Handles gPlatformHasDeviceTreeEvent, installing DT as a configuration table
  - Does not implement FDT_CLIENT_PROTOCOL.
Let’s See That Example Again

genet: ethernet@7d580000 {
    compatible = "bcm,bcm2711-genet-v5";
    reg = <0x0 0x7d580000 0x10000>;
    address-cells = <0x1>;
    size-cells = <0x1>;
    interrupts = <GIC_SPI 157 IRQ_TYPE_LEVEL_HIGH>,
                 <GIC_SPI 158 IRQ_TYPE_LEVEL_HIGH>;
    status = "disabled";

    phy-handle = <&phy1>;
    phy-mode = "rgmii-rxid";
    status = "okay";

    genet_mdio: mdio@e14 {
        compatible = "bcm,genet-mdio-v5";
        reg = <0xe14 0x8>;
        reg-names = "mdio";
        address-cells = <0x1>;
        size-cells = <0x0>;

        phy1: ethernet-phy@1 {
            /* NO PHY interrupt */
            reg = <0x1>;
        };
    };
};
Property Lookup

- Actual properties can be an array of the same type, or even compound.
- API centered around sequential parsing.
- EFI_DT_IO_PROTOCOL_GET_PROP returns an EFI_DT_PROPERTY
  - VOID *Begin
  - VOID *Iter
  - VOID *End
- EFI_DT_IO_PROTOCOL_PARSE_PROP takes Type, Index and advances Iter.
- Convenience wrappers (GetReg)
## Property Lookup

<table>
<thead>
<tr>
<th>EFI_DT_VALUE_TYPE</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>EFI_DT_VALUE_U32</td>
<td>A 32-bit value.</td>
</tr>
<tr>
<td>EFI_DT_VALUE_U64</td>
<td>A 64-bit value.</td>
</tr>
<tr>
<td>EFI_DT_VALUE_BUS_ADDRESS</td>
<td>An address encoded by #address-cells.</td>
</tr>
<tr>
<td>EFI_DT_VALUE_SIZE</td>
<td>A size encoded by #size-cells.</td>
</tr>
<tr>
<td>EFI_DT_VALUE_REG</td>
<td>A reg property value.</td>
</tr>
<tr>
<td>EFI_DT_VALUE_RANGE</td>
<td>A ranges/dma-ranges property value.</td>
</tr>
<tr>
<td>EFI_DT_VALUE_STRING</td>
<td>A string property value.</td>
</tr>
<tr>
<td>EFI_DT_VALUE_LOOKUP</td>
<td>A reference to another DT device (EFI_DT_IO_PROTOCOL).</td>
</tr>
</tbody>
</table>
Parsing can be complicated

- The number of cells in a bus address `reg` is controlled by parent device `#address-cells` and `#size-cells`
- Limit on address/size values is `__int128` on 64-bit systems.
- “reg” may be translated.

Reg[0] is [0x4600, 0x4700), and within parent’s [0xe0000000,0xe0100000)

Reg[0] is thus translated to [0xe004600, 0xe004700), can can be accessed using I/O accessors for the `soc` node.

If something translates all the way to root node, it’s in CPU address space.
I/O Access

- ReadReg/WriteReg/PollReg.
- Can be hooked by drivers.

```c
// This matches the CpuIo2 EFI_CPU_IO_PROTOCOL_WIDTH.
// When we go 128-bit, this will need work.

typedef enum {
    EfiDtiIoWidthUint8  = 0,
    EfiDtiIoWidthUint16,
    EfiDtiIoWidthUint32,
    EfiDtiIoWidthUint64,
    EfiDtiIoWidthFifoUint8,
    EfiDtiIoWidthFifoUint16,
    EfiDtiIoWidthFifoUint32,
    EfiDtiIoWidthFifoUint64,
    EfiDtiIoWidthFillUint8,
    EfiDtiIoWidthFillUint16,
    EfiDtiIoWidthFillUint32,
    EfiDtiIoWidthFillUint64,
    EfiDtiIoWidthMaximum
} EFI_DT_IO_PROTOCOL_WIDTH;
```
DMA Buffers

Buffer handling can be complicated.

- Need to honor **dma-ranges**.
  - Device may have restricted I/O capabilities.
  - Translation between a device bus address and the CPU view of the buffer.
- Need to honor **dma-coherent**
  - May imply MMU attributes
  - May imply bounce buffering
- There can be also CPU barriers hidden by the Map/Unmap API.
Two Usage Patterns

• Driver Binding

• Legacy (Like FdtClientDxe, manual scraping of tree nodes):
  – Well suited to lib-based drivers, e.g. using:
    • SerialPortLib
    • PciHostBridgeLib
HighMemDxe – Legacy Way

- DEPEX on gEfiDtIoProtocolGuid (FdtBusDxe loading)
- InitializeHighMemDxe:
  - LocateHandleBuffer (gEfiDtIoProtocolGuid)
  - For Handle in HandleBuffer:
    - If AsciiStrCmp (DtIo->DeviceType, "memory") != 0 && DtIo->DeviceStatus == EFI_DT_STATUS_OKAY
      - ProcessMemoryRanges
HighMemDxe – Driver Binding

- InitializeHighMemDxe:
  - EfiLibInstallDriverBindingComponentName2

- DriverSupported
  - AsciiStrCmp (DtIo->DeviceType, "memory") != 0 && DtIo->DeviceStatus == EFI_DT_STATUS_OKAY

- DriverStart
  - ProcessMemoryRanges
HighMemDxe – ProcessMemoryRanges

Index = 0;
do {
    Status = DtIo->GetReg (DtIo, Index++, &Reg);
    if (EFI_ERROR (Status)) {
        if (Status != EFI_NOT_FOUND) {
            DEBUG ((DEBUG_ERROR, "%a: GetReg(%a): \%r\n", __func__, DtIo->Name, Status));
        } else {
            Status = EFI_SUCCESS;
        }
        break;
    }

    if (Reg.BusDtiO != NULL) {
        DEBUG ((DEBUG_ERROR, "%a: range 0x%x - 0x%x are not CPU real addresses\n", __func__, Reg.Base, Reg.Base + Reg.Length - 1));
        Status = EFI_UNSUPPORTED;
        break;
    }

    Status = ProcessMemoryRange (&Reg);
    if (EFI_ERROR (Status)) {
        DEBUG ((DEBUG_ERROR, "%a: ProcessMemoryRange(%a): \%r\n", __func__, DtIo->Name, Status));
        break;
    }
} while (1);
HighMemDxe – Driver Binding

...might wonder what forces the binding to happen at boot.

• This could be via Bds, similarly as to how video devices are connected even on a boot without full enumeration.

• Some node are marked as critical nodes – must be connected.
  – Nodes of type `memory`
  – Nodes with `uefi,critical` property present
Thanks for attending the UEFI Fall 2023 Developers Conference & Plugfest

For more information on UEFI Forum and UEFI Specifications, visit http://www.uefi.org