*presented by*


# Virtualization for UEFI DXE Driver Testing and Deployment Validation

## UEFI 2025 Developers Conference & Plugfest

9 October 2025

Aaron Rossetto

# Meet the Presenter

## Aaron Rossetto

Windows Device Driver Engineer

Austin, Texas, USA

National Instruments (now Emerson) 2000-2022

- Drivers and firmware for high-speed data acquisition and software-defined radio devices

Cirrus Logic, 2022-

- Drivers for PC audio silicon

Specialties include low-level Windows and Linux system software and firmware development, debugging and deployment

# Agenda

- Motivation
- Workflow and Problem
- Solution
- System Overview
- Results and Conclusions

# Motivation

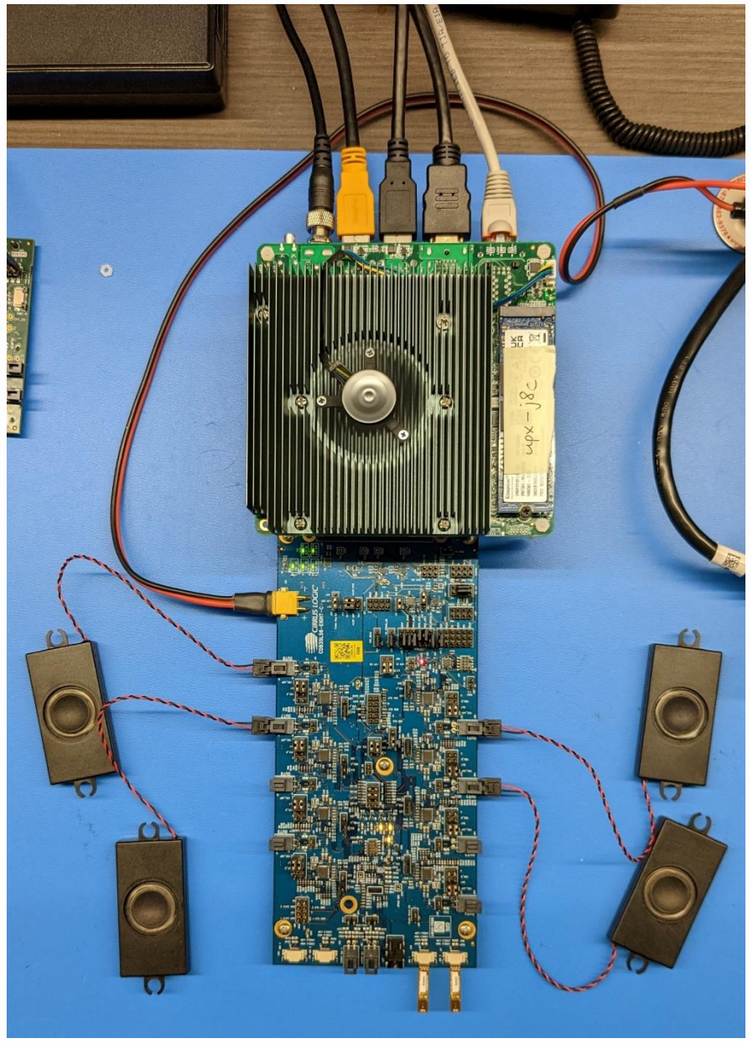
- Story begins on day one of my employment at Cirrus Logic

# Motivation

- Customer requirement: Firmware and configuration download to onboard peripheral from secure environment
- “They’re asking us about Driver Execution Environment (DXE) drivers”
- My job: *Go figure it out*

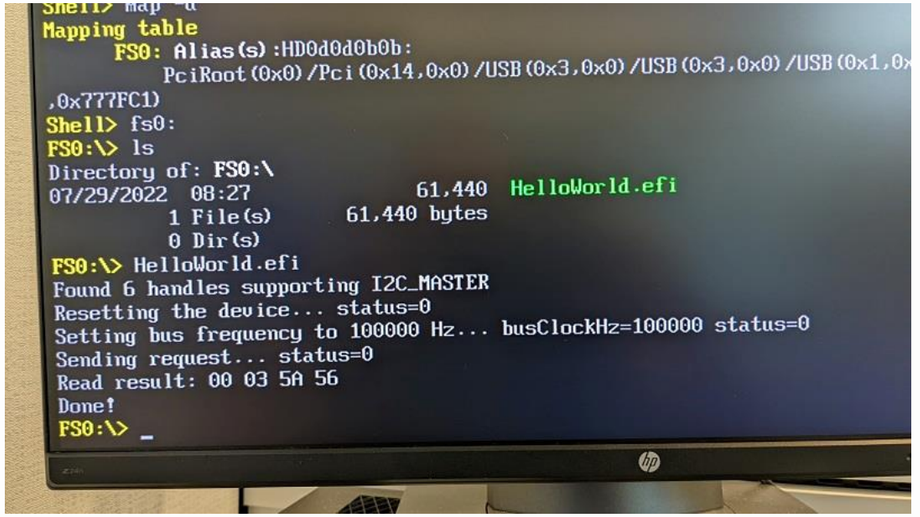
# Workflow and Problem

- Commercial dev platform with custom silicon dev board on Hardware Attached on Top (HAT) connector
- $I^2C$ and Mobile Industry Processor Interface (MIPI) SoundWire interfaces to silicon
- Step 1: Talk to the silicon



# Workflow and Problem

- No UEFI `EFI_I2C*` protocols in stock BIOS
- Wrote I²C controller driver based on older SoC code in EDK2-`platforms` repo
- Success!



# Workflow and Problem

- Load $I^2C$ controller driver manually from shell for `EFI_I2C_MASTER_PROTOCOL`
- Load firmware update driver from shell
  - Read HW configuration from EFI variable
  - Read firmware files from EFI system volume
  - Program firmware via $I^2C$ driver

# Workflow and Problem

- Great! Job well done!
- ...until boss asks, *“How does this work on a customer system?”*

# Workflow and Problem

- Workflow great for manual testing, but not representative of operation on a customer system
  - Firmware should come from Firmware Volume (FV), not EFI system volume
  - Config should be in PCD, not EFI variable
  - Customers shouldn’t need to manually boot to the UEFI shell to initialize the silicon!

# Workflow and Problem

- But building a custom BIOS is not an option
- “We are the integrator”: Customers expect us to know how to integrate the solution on their platforms
- *What do I do?*

# Solution

- OVMF – Open Virtual Machine Framework
  - OVMF is an EDK II based project to enable UEFI support for virtual machines
  - OVMF contains sample UEFI firmware for Quick EMUlator (QEMU) and Kernal-based Virtual Machine (KVM)

# Solution

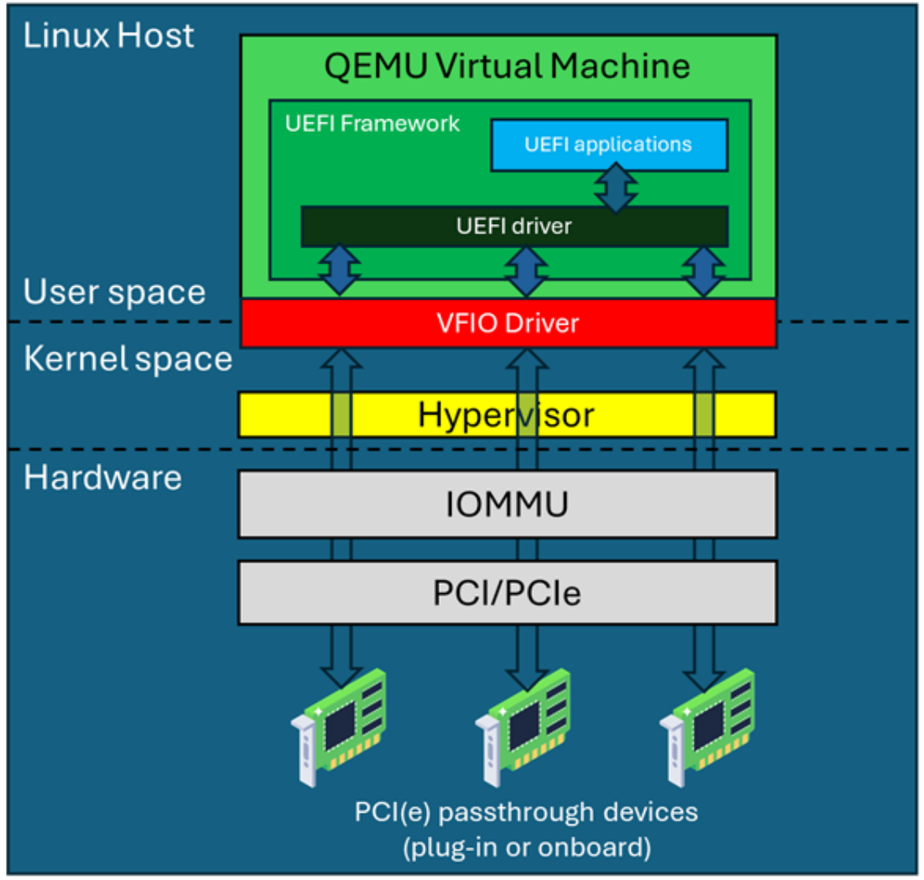
- Good option for learning how to do things the “right way”
  - Integrating DXE driver binary into a UEFI BIOS
  - Integrating files into firmware volume
  - Updating platform config data for driver
- What about talking to the silicon?

# Solution

- $I^2C$ controller is in SoC, accessible via PCI
- PCI passthrough commonly used with QEMU to pass through GPUs to the guest
- Idea: Use PCI passthrough to hoist $I^2C$ controller PCI device into QEMU and make accessible to OVMF

# System Overview

- Ubuntu 22.04 on dev platform (host)
- QEMU running local OVMF build
- PCI passthrough of $I^2C$ controllers enabled in kernel options



# System Overview

- PCI passthrough requires all devices in same Input-Output Memory Management Unit (IOMMU) group to be passed through to the VM
- `lspci --nvv` shows IOMMU group of PCI devices
- `8086:a0c5` (PCI address 0:19.0) is PCI vendor/device ID of $I^2C$ controller to pass through

```
00:19.0 0c80: 8086:a0c5 (rev 20)
        DeviceName: Onboard - Other
        Subsystem: 8086:7270
        ... other output removed ...
        IOMMU group: 10

00:19.2 0780: 8086:a0c7 (rev 20)
        DeviceName: Onboard - Other
        Subsystem: 8086:7270
        ... other output removed ...
        IOMMU group: 10
```

# System Overview

- Kernel options updated to enable IOMMU and bind *all* group 10 devices to VFIO (virtual function I/O) driver

```
intel_iommu=on iommu=pt vfio-pci.ids=8086:A0C5,8085:A0C7
vfio-pci.disable_idle_d3=on
```

# System Overview

- `lspci -kn` confirmed attachment of PCI devices to `vfio-pci` driver

```
00:19.0 0c80: 8086:a0c5 (rev 20)
        DeviceName: Onboard - Other
        Subsystem: 8086:7270
        Kernel driver in use: vfio-pci
        Kernel modules: intel_lpss_pci
00:19.2 0780: 8086:a0c7 (rev 20)
        DeviceName: Onboard - Other
        Subsystem: 8086:7270
        Kernel driver in use: vfio-pci
        Kernel modules: intel_lpss_pci
```

# System Overview

- QEMU started specifying `vfio-pci` device and I2C controller PCI address and custom OVMF BIOS build location

```
user@host:~/uefi/edk2$ sudo qemu-system-x86_64 -accel kvm \
   --bios Build/OvmfX64/RELEASE_GCC5/FV/OVMF.fd \
   -net none -machine q35 \
   -device vfio-pci,host=00:19.0,multifunction=on,x-no-mmap=on
```

# Results and Conclusions

- `pci` UEFI shell command confirms presence of passthrough PCI device
- Note the different PCI address

```
UEFI Interactive Shell v2.2
EDK II
UEFI v2.70 (EDK II, 0x00010000)
Mapping table
     BLK0: Alias(s):
          PciRoot(0x0)/Pci(0x1F,0x2)/Sata(0x2,0xFFFF,0x0)
Press ESC in 1 seconds to skip startup.nsh or any other key to continue.
Shell> pci
   Seg  Bus  Dev  Func
   ---  ---  ---  ----
    00   00   00    00 ==> Bridge Device - Host/PCI bridge
                Vendor 8086 Device 29C0 Prog Interface 0
    00   00   01    00 ==> Display Controller - VGA/8514 controller
                Vendor 1234 Device 1111 Prog Interface 0
    00   00   02    00 ==> Serial Bus Controllers - Other bus type
                Vendor 8086 Device A0C5 Prog Interface 0
    00   00   1F    00 ==> Bridge Device - PCI/ISA bridge
                Vendor 8086 Device 2918 Prog Interface 0
    00   00   1F    02 ==> Mass Storage Controller - Serial ATA controller
                Vendor 8086 Device 2922 Prog Interface 1
    00   00   1F    03 ==> Serial Bus Controllers - System Management Bus
                Vendor 8086 Device 2930 Prog Interface 0
Shell> _
```

# Results and Conclusions

- Driver loaded manually and diagnostic utilities confirm attachment of EFI_I2C_MASTER_PROTOCOL and successful communication with hardware

```
Shell> fs0:
FS0:\> load -nc SimpleI2cPortDxe.efi
Image 'FS0:\SimpleI2cPortDxe.efi' loaded at 66F6000 - Success
FS0:\> _
```

```
FS0:\> dh -p I2cMaster
Handle dump by protocol 'I2cMaster'
95: I2cMaster PCIIO DevicePath(PciRoot(0x0)/Pci(0x2,0x0))
```

```
FS0:\> I2cMasterTest -i 0x95 -r 0 1 -d 0x32
dev 50
Resetting the device... ok!
Setting bus speed to 100000 Hz... ok! Bus speed is 100000 Hz
Read address: 00000000
Sending request... ok!
Read result:
00000000: 00035A56
Done! (Success)
```

# Results and Conclusions

- QEMU and OVMF are great alternatives when building a custom BIOS is not an option
- Additional benefit of QEMU: monitor mode to examine state of VM without external debugger
  - Invaluable for register-level access debugging

# Results and Conclusions

- Enable on command line with –monitor option

```
user@host:~/uefi/edk2$ sudo qemu-system-x86_64 -accel kvm \
   --bios Build/OvmfX64/RELEASE_GCC5/FV/OVMF.fd \
   -net none -machine q35 \
   -device vfio-pci,host=00:19.0,multifunction=on,x-no-mmap=on \
   -monitor stdio
```

- At (qemu) prompt: log trace:vfio*

# Results and Conclusions

- Caution: QEMU appears to have a 64 kB limitation with custom DSDT

```
libvirt.libvirtError: internal error: process exited while
connecting to monitor: qemu-system-x86_64: -acpitable
file=dsdt.bin: ACPI table too big, requested: 283067, max: 65535
```

  - May prove problematic for devices whose drivers rely on ACPI data

# Results and Conclusions

- Powerful technique… but nothing can beat real hardware
- Do readily available platforms with easy-to-customize BIOSes exist?
- More options for non-IBVs or ODMs to build custom or customize their BIOS

# Questions?

# References

- QEMU: https://www.qemu.org/
- OVMF: https://github.com/tianocore/tianocore.github.io/wiki/OVMF
- Configuring Linux kernel for PCI passthrough: https://docs.redhat.com/en/documentation/red_hat_virtualization/4.1/html/installation_guide/appe-configuring_a_hypervisor_host_for_pci_passthrough