presented by

How writing portable UEFI drivers improves reliability (and helps me)

Spring 2019 UEFI Plugfest
April 8-12, 2019
Presented by Leif Lindholm & Ard Biesheuvel (Linaro)

Agenda

- Architectures vs. platforms
- UEFI on Arm
- Practical differences
- Testing on other platforms
- Final Notes

Architectures vs. platforms

What is in an architecture?

Porting drivers to other architectures is not just a question of instruction set.

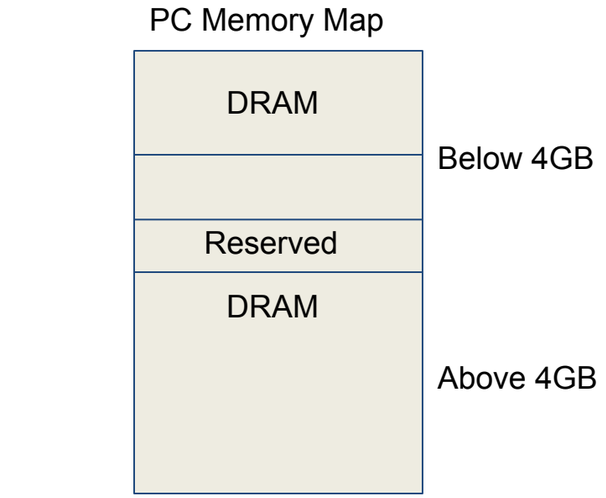
UEFI defines fully portable interfaces, but “x86” (Ia32/X64) will let you get away with shortcuts that cannot be relied on for Arm systems.

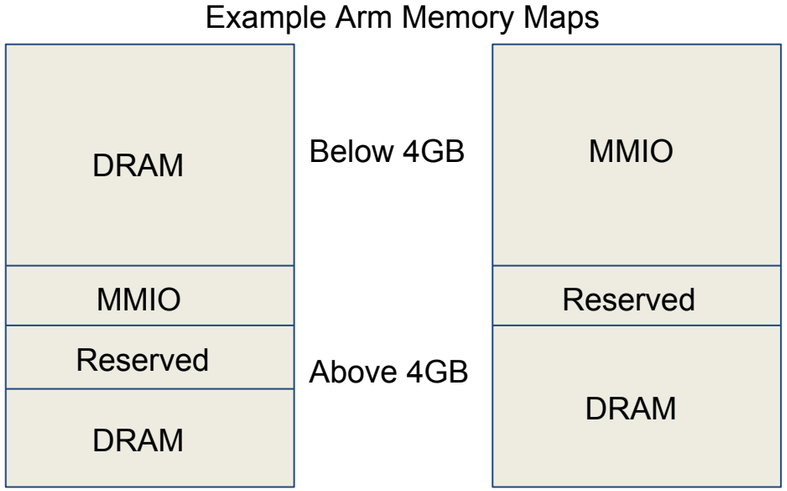
When talking about “x86”, people generally mean a descendant of the “IBM PC compatible” platform. Arm (ARM/AARCH64) does not have any such rigidly defined platform.

Arm Memory organization

The Arm architecture does not standardize the CPU's view of the physical address space

- DRAM may start at address 0x0, but often it does not
- 'Standard' Arm peripherals such as the GIC (Generic Interrupt Controller) could be placed anywhere
- Traversing DRAM to find ACPI or SMBIOS tables by magic numbers is **not** possible
  - Linux's `/dev/mem` should **not** be used in production code





UEFI on Arm

UEFI on Arm - background

Bindings for AARCH64 included in UEFI 2.4, released June 2013.

Initial support merged in upstream EDK2 18th July 2013.

In UEFI, 32-bit Arm is known as **ARM**. 64-bit Arm is known as **AARCH64**.

For practical purposes, no processors older than Arm Architecture v7 can implement support for UEFI. For data centres, we only care about AArch64.

Arm and UEFI

Arm Partners license processors, or the architecture, and develop their own silicon. This enables a lot of innovation, but it also risks leading to fragmentation.

UEFI is part of the solution for how to reduce this fragmentation. Various companion specifications describe how to create systems that can be supported by UEFI. By following those specifications, OEMs can create systems that will work like customers expect. By following UEFI strictly, others can create drivers and applications that will work across these different systems.

Linaro helps to enable the ecosystem, by giving a natural place for Arm licensees (who are competitors) to work together for mutual benefit.

Differentiation or being different?

On its own, the Arm architecture does not contain enough defined behaviour to enable off-the-shelf operating systems to run.

On the data centre side, this has been managed by the introduction of two specifications from Arm:

- Server Base System Architecture (SBSA)
  - Describes what the hardware must conform to
- Server Base Boot Requirements (SBBR)
  - Describes how the firmware must behave

Efforts are underway to do the same for the embedded segment via the introduction of an Embedded BBR - EBBR.

Practical differences

Option ROM emulation

Ard Biesheuvel (Linaro) and Alex Graf (SuSe) put together a solution to execute X64 option ROMs on AARCH64[1] via emulation.

The problem with emulation is that you need to mimic bug behaviour as well as correct behaviour. The option ROM emulation work turned up some common issues when testing drivers on AARCH64:

- Failure to check for NULL pointers leading to spurious memory accesses near address 0.
- Assumptions that DMA or other buffers need to be located < 4GB.
- Failure to use proper UEFI interfaces for PCI/DMA.

None of the above would trigger an error on a PC platform.

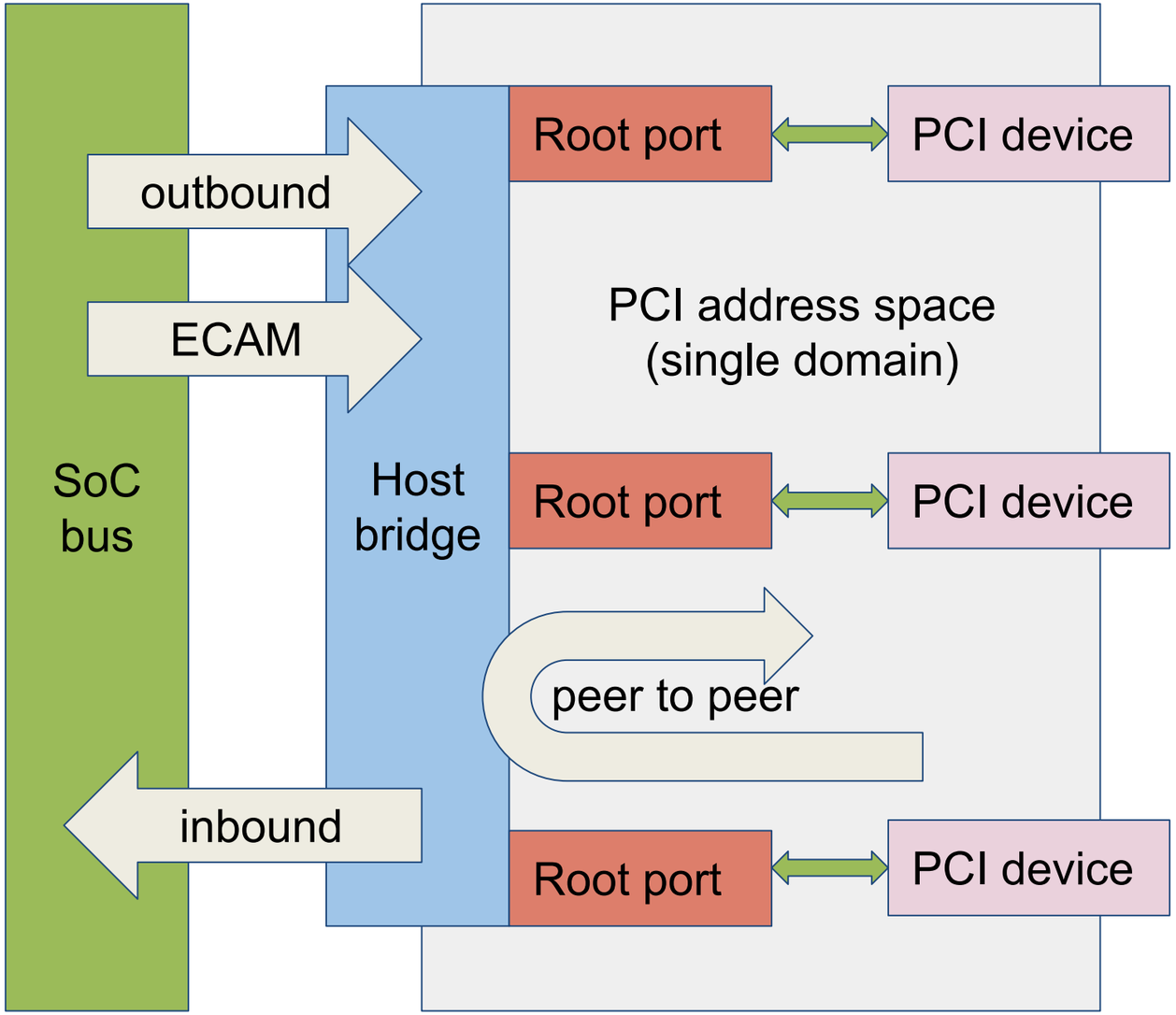
[1] Presentation given at KVM forum 2017: https://www.youtube.com/watch?v=uxvAH1Q4Mx0

PCIe on Arm

The Arm architecture does not cover PCIe memory organization or topology, so anything that the PCIe specification permits could potentially be found in an Arm system:

- Outbound translation
- Inbound translation
- Non-cache coherent DMA (although not permitted by SBSA)
- Single outbound MMIO window (for 32-bit and 64-bit prefetchable and non-prefetchable BARs)
- No outbound I/O window
- Multiple ECAM config windows could be located anywhere in the CPU address space

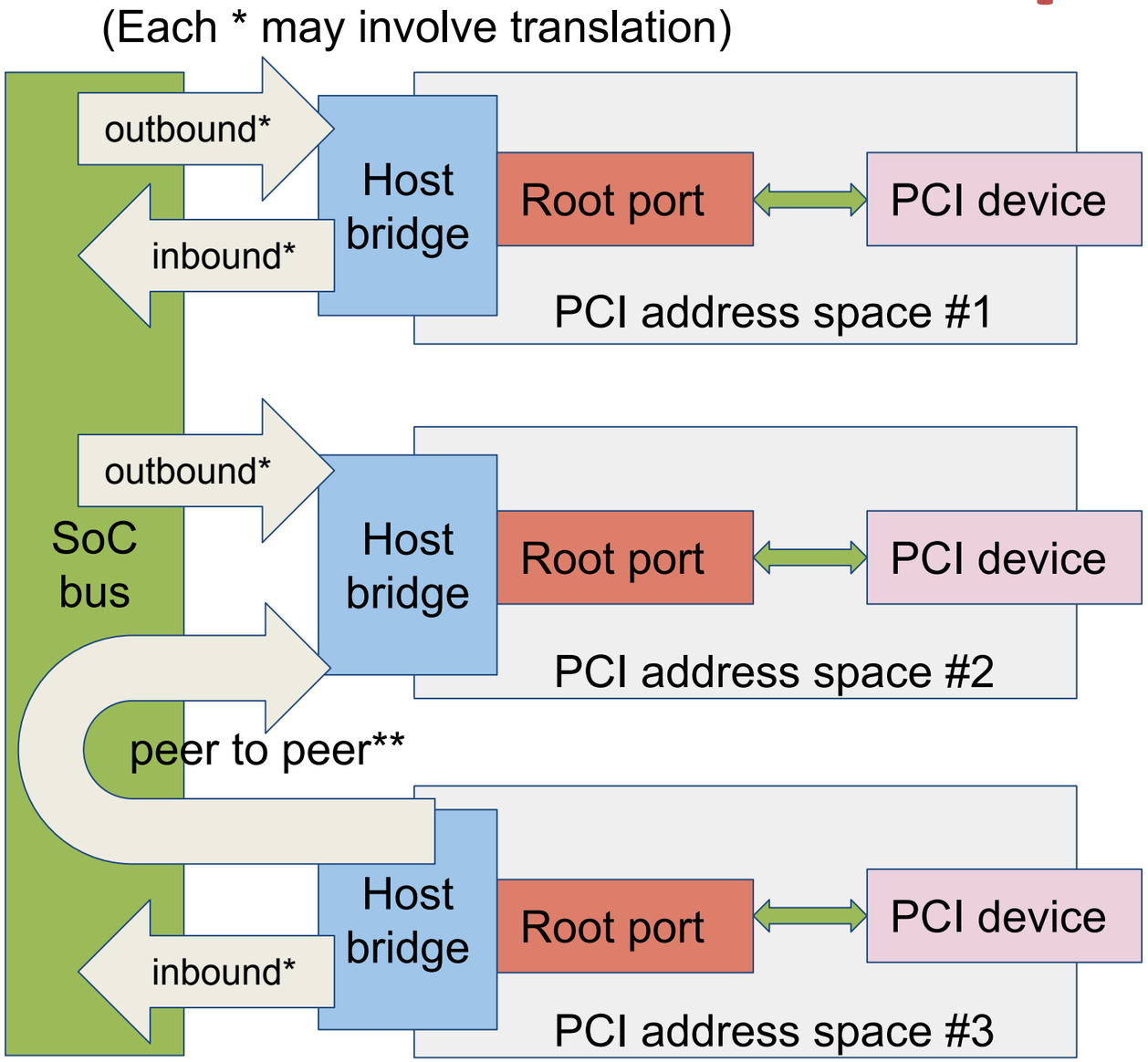
Typical PCIe implementation (PC)



- All root ports are behind a single host bridge.
- Outbound transactions (BAR accesses) are mapped 1:1
- Inbound transactions (DMA accesses) are mapped 1:1
- Peer-to-peer accesses are local to the PCI address space

All memory transactions occur in the same address space

Permitted PCIe implementation



- One host bridge per slot (and sometimes the root port PPB logic is omitted entirely)
- Outbound transactions (BAR accesses) may be translated, to support 32-bit BARs while preserving CPU address space
- Inbound transactions (DMA accesses) may be translated, to make DRAM accessible to masters that are only 32-bit DMA capable
- Peer-to-peer accesses may be translated twice (!)

Portable drivers for PCI devices

The PCI protocols in the UEFI spec were carefully designed to abstract away from platform specific properties such as translation.

- `Map()` and `Unmap()` methods deal with inbound translation and non-cache coherent DMA
- `AllocateBuffer()` deals with non-cache coherent DMA as well
- `GetBarAttributes()` deals with outbound translation (i.e., it returns the BAR value translated for the CPU)

- Unfortunately, drivers often cut corners, violating the UEFI spec
- Many such drivers have only ever been tested on x86/PC, where they do not cause unexpected behavior
- On more exotic PCIe topologies, such drivers will **not** work

Portable code (EDK2 EHCI driver)

```
 Status = PciIo->AllocateBuffer (PciIo, AllocateAnyPages, EfiBootServicesData, Pages, &BufHost, 0);
  if (EFI_ERROR (Status)) {
    goto FREE_BITARRAY;
  }

  Bytes = EFI_PAGES_TO_SIZE (Pages);
  Status = PciIo->Map (PciIo, EfiPciIoOperationBusMasterCommonBuffer, BufHost, &Bytes, &MappedAddr,
&Mapping);
  if (EFI_ERROR (Status) || (Bytes != EFI_PAGES_TO_SIZE (Pages))) {
    goto FREE_BUFFER;
  }

  ...

  Block->BufHost  = BufHost;                                  // CPU address
  Block->Buf      = (UINT8 *) ((UINTN) MappedAddr);           // device address (could be != CPU address)
  Block->Mapping  = Mapping;
```

Portable code - description

This is a fairly straight-forward example of using UEFI's PCI DMA API, but there a few things to note:

- `PciIo->Map()` can only called with the `EfiPciIoOperationBusMasterCommonBuffer` mapping type if the memory was allocated using `PciIo->AllocateBuffer()`.
- the physical address set in `MappedAddr` by `PciIo->Map()` can deviate from both the virtual and physical addresses as seen by the CPU (note that UEFI maps system memory VA to PA 1:1).
- the size of the actual mapping can deviate from the requested size.

Works on a PC - not portable

On a PC, PCI is cache coherent and 1:1 mapped. So the following code will work just as well:

```
Status = gBS->AllocatePages (AllocateAnyPages, EfiBootServicesData, Pages, &BufHost);
  if (EFI_ERROR (Status)) {
    goto FREE_BITARRAY;
  }

  ...

  Block->BufHost  = BufHost;              // CPU address
  Block->Buf      = BufHost;                  // device address (assumed to be equal to CPU
address)
```

The following slides give some examples of how a non-PC system can deviate from a PC when it comes to its physical address space layout.

On a PC, DRAM starts at 0x0

Taken from the VLogError() routine in SHIM:

```
size = SPrint(NULL, 0, L"%a:%d %a() ", file, line, func);
newerrs[nerrs] = AllocatePool(size*2+2);
SPrint(newerrs[nerrs], size*2+2, L"%a:%d %a() ", file, line, func);
```

On a PC, DRAM starts at 0x0 (2)

On a PC, DRAM starts at address 0x0, and most of the 32-bit addressable physical region is used for memory.

This has multiple implications for software:

- inadvertent NULL pointer dereferences from UEFI code may go entirely unnoticed (seen in option ROMs on a common graphics card)
- PCI devices that only support 32-bit DMA (or need a little kick to support more than that) can expect to always be able to access buffers at a 1:1 mapping
  - most UEFI implementations for PC explicitly limit PCI DMA to 4 GB
  - most UEFI PCI drivers do not set the EFI_PCI_IO_ATTRIBUTE_DUAL_ADDRESS_CYCLE attribute, required for >32 bit DMA capable hardware

On a PC, DRAM starts at 0x0 (3)

- Much existing code incorrectly verifies that pointers and buffers reside fully in 32-bit space, signalling an error condition when this is not the case.

On Arm systems, the amount of available 32-bit addressable RAM may be much smaller, or it may even be absent entirely.

In the latter case, hardware that is only 32-bit DMA capable can only work if

- an IOMMU is present and wired into the PCI root bridge driver by the platform

or

- if translation is applied to inbound PCIe transactions.

On a PC, DRAM starts at 0x0 (4)

But in general, it should be expected that Arm platforms use at least 40 bits of address space for DMA, and so drivers for 64-bit DMA capable peripherals **must** enable this capability in the hardware.

Also - some code only tested on PC assumes that addresses *just below* 4GB cannot be DRAM.

- Potential arithmetic errors on 32-bit *or* if using 32-bit arithmetic (intentionally or not).

On a PC, DMA is cache coherent

Although not that common, it is possible and permitted by the UEFI spec for PCI DMA to be non cache coherent. This is completely transparent to the driver, **provided that it uses the APIs correctly**.

For instance, `PciIo->AllocateBuffer()` will return an uncached buffer in this case, and the `Map()` and `Unmap()` methods will perform cache maintenance under the hood to keep the CPU's and the device's view of memory in sync.

PC has strongly ordered memory

UEFI is mostly a single-threaded execution environment, and if the PI `MP_PROTOCOL*` is in use, explicit synchronization mechanisms are provided.
Arm only guarantees the ordering of memory accesses in regions marked as Device or Strongly-ordered. And even there, they are only guaranteed within the same 1kB aligned 1kB region.
If a device driver requires two memory accesses to take place in a specific order, MemoryFence() must be used.

- update DMA descriptor, initiate transaction
- Write, delay, poll

Likewise, any delay timeout after a write to a memory mapped register probably needs a `MemoryFence()`.

Testing on other platforms

Wanna play?

Ard has put together a worst-case system, nicknamed ‘Chucky’, for triggering the most common issues in drivers.

The system has two separate PCIe host bridges, and several selectable behaviors configurable by DIP switches:

- MMIO translation for host bridge A, host bridge B uses 1:1 mappings for MMIO
- DRAM below 4GB can be remapped to a CPU address above 4 GB
  - No 32-bit DMA on host bridge A
  - Inbound translation for 32-bit DMA on host bridge B
- Setting of Bus Master Enable bit can be deferred to first PCI I/O Map() call
- No DRAM near zero to catch NULL pointer dereferences.

Cross-architecture toolchains

Visual Studio support for AArch64 available since VS2017 15.4

Majority of open-source Arm development uses GCC

LLVM supported, both in form of clang and Xcode

Final notes

Overall message

It is completely possible to write a driver or application that can be easily compiled for either X64 or AARCH64 architectures.

It is also very much possible to write a driver that works on X64 and not on AARCH64 (but it is usually due to bugs).

Writing drivers portably increases code correctness and reliability. Testing on multiple architectures is even better :)

Questions?

Thanks for attending the 2019 Spring UEFI Plugfest

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

presented by

Leif Lindholm & Ard Biesheuvel

leif.lindholm@linaro.org
ard.biesheuvel@linaro.org

Using encrypted memory for DMA

This case is actually rather similar to the non cache coherent DMA case, in the sense that the *allocate*, *map* and *unmap* actions all involve some extra work performed by the platform under the hood. This implies that non-compliant drivers will not work on x86 systems with encrypted memory either.

Common DMA buffers are allocated from unencrypted memory, and mapping or unmapping involve decryption or encryption in place depending on the direction of the transfer (or bounce buffering if encryption in place is not possible, in which case the device address will deviate from the host address like in the non-1:1 mapped PCI case above). Cutting corners here means that attempted DMA transfers will produce corrupt data, usually a strong motivator to get your code fixed.