WHITE PAPER

Device Virtualization Principles for Real-time Systems

Arm v8-R Device Virtualization

Paul Hughes
Lead System Architect and Distinguished Engineer ATG

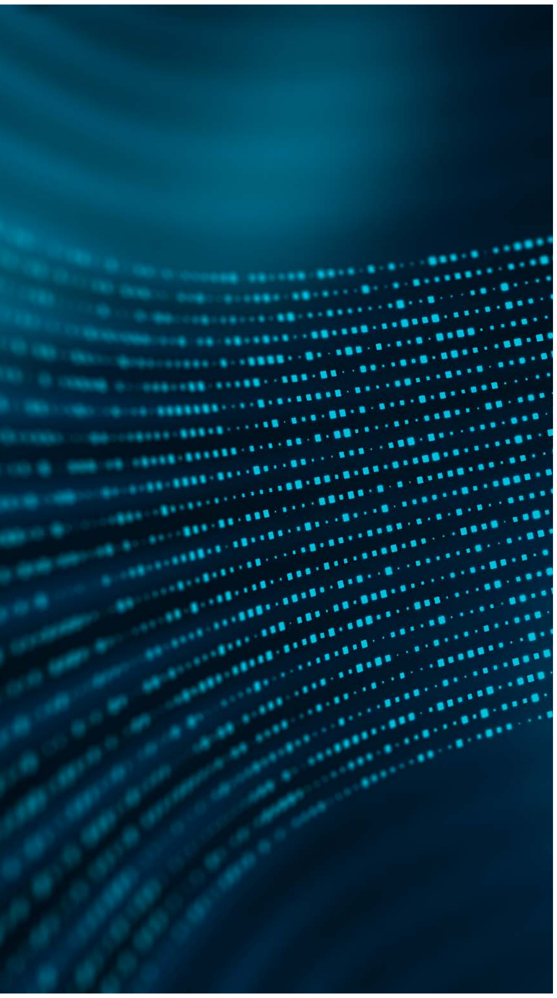
Bernhard Rill
Director Automotive Partnerships EMEA

Alexandre Romana
System Architect ATG

James Scobie
Director Automotive Product Management

Device Virtualization Principles for Real-time Systems

Arm v8-R Device Virtualization

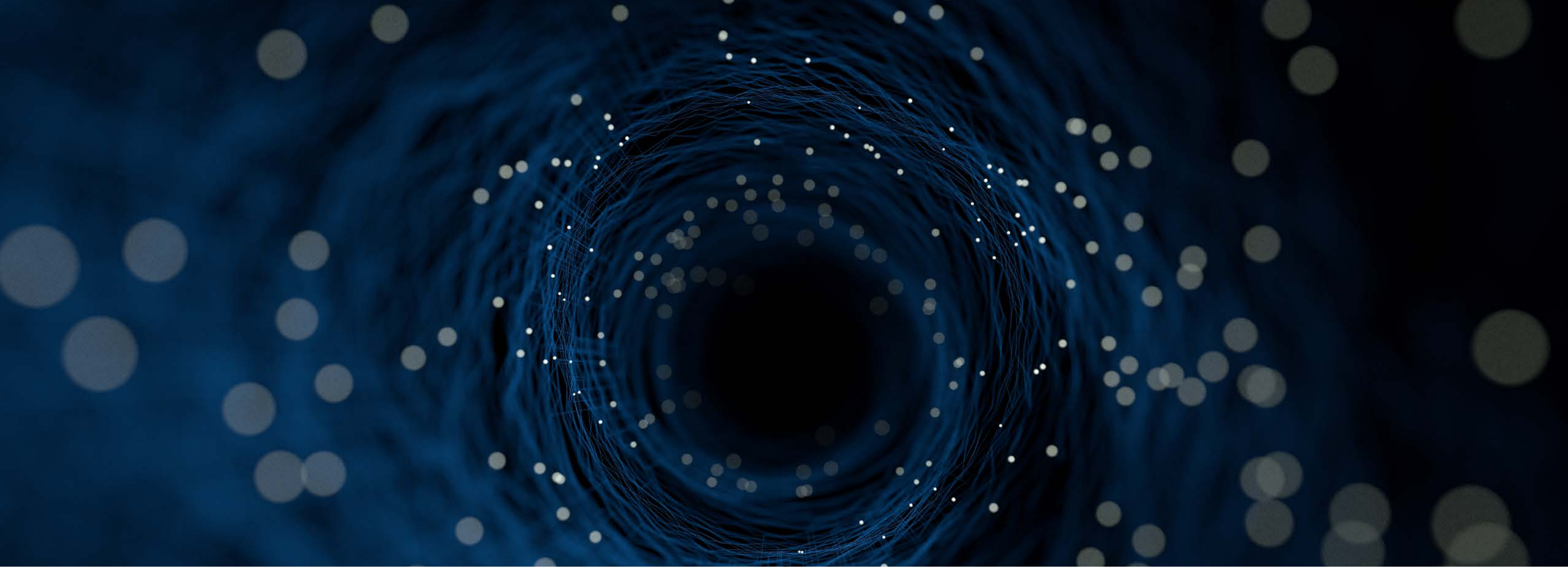


SUMMARY

Virtualization is now at the heart of a revolution that is taking place in the domains served by the Armv8-R architecture. For example, the EL2 separation option in Cortex-R52+ represents a good option to enable intelligent integration of multiple software stacks (more details can be found in the Arm Best Practices for Armv8-R Cortex-R52+ Software Consolidation paper). While the CPU architecture has evolved to provide such features enabling this virtualization, on the device side (whether hardware accelerators or I/O peripherals) implementing a proper isolation for safety and security while balancing performance and cost may prove challenging. Depending on use cases, multiple solutions with different software and hardware cost versus efficiency ratios will prove optimal. This whitepaper discusses those approaches to provide guidance, as a prerequisite for the proper standardization of device virtualization in real-time systems.

Contents

Non-Confidential Proprietary Notice



01 Introduction

Virtualization is no longer exclusive to cloud servers or even personal computers. Today, it is being deployed everywhere. Since Arm is leveraging the same technologies from cloud to real-time devices, it is very well positioned to unlock breakthrough use cases and help this revolution.

Embedded virtualization has become one of the key technologies to handle the complexity of functionalities in a wide range of applications. For example, the automotive industry is exploring autonomous driving, enhanced connectivity, and disruptive mobility, wherein the role of software development is predicted to increase exponentially.

With this increase in software complexity and the number of components in those computers on wheels, virtualization is a vital technique to ease software integration from different suppliers into a domain or zonal controller.

However, while it comes with those many benefits, it also adds some complexity to device use model:

- How do you allocate device resources to multiple safely and securely isolated applications and operating systems?
- How do you scale the resources?
- How do you maintain the principle of least privilege?

There are currently two ways to make a device available to software:

- Mediation
  - by the OS: System calls.
  - by the emulation layer: Emulation.
  - by the hypervisor: Paravirtualization.
- Assignment

Device mediation has been predominant so far and addresses a wide variety of use cases, for example for low performance devices or when device accesses are infrequent. However, this does not scale well and may negatively impact performance as well as power consumption. For those use cases, device assignment (where a physical device is directly accessible from within virtualized software) may prove a superior solution. Notably, it:

- Provides near-native device performance.
- Minimizes trusted computing base (TCB) footprint by removing device specific code.
- Reduces upstream cost/complexity.
- Enables driver domains/driver virtual machines (VM).
- Enables user space drivers, with all its benefits, such as closed source drivers and choice of programming language for those drivers, to name a few.

To take full advantage of these benefits, systems and devices must be designed in a way that avoids current pitfalls around device assignment: device scaling and isolation. Indeed, multi-tenant systems must be able to share those devices while maintaining isolation and real time.

This whitepaper aims at providing a set of requirements to enable the secure assignment of portions of an on-chip device to a software entity running on an Arm v8-R CPU. Its goal is to unlock full devices resource utilization in virtualized environments, and to enable a proper device partitioning in such systems.

02 Device Virtualization Design Patterns

As device sharing is becoming ubiquitous in virtualized systems (on Armv8-A based systems and increasingly on Armv8-R systems), we have seen a number of device virtualization patterns emerging.

In the diagrams below, we describe some of the most commonly used patterns:

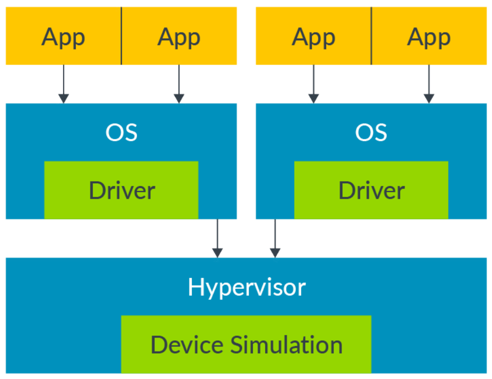
- Device virtualization
- Device emulation
- Device paravirtualization
- Device pass-through
- Driver VM
- Mediated Pass-through
- Standard multi-context device pass-through

We use OS and VM interchangeably, and blue arrows depict memory sharing between the device context and the software agent. Interrupts are not depicted as they are often flexible, from multiple redirections to direct injection.

Device Simulation

In some cases, a device that is needed by a virtualized software is not physically available. In such case, where possible, a solution can be to fully emulate the device functionality with software running in another VM or in the hypervisor. This solution is the one with the most limitations.

FIG. 1
Device Simulation example

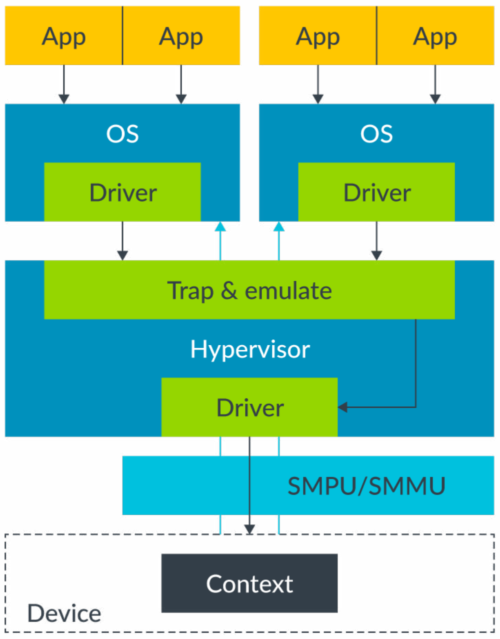


Device Emulation

Traditionally, devices have been virtualized by trapping and emulating device control and data path in the hypervisor. When the device is shared, the driver in the hypervisor or the emulation layer must arbitrate between VMs and avoid interference between requestors; for example, using time-sharing. This design, however, may introduce a performance cost and requires the hypervisor to include a device specific driver, increasing safety, security and certification related challenges. For example, UART is often virtualized this way.

FIG. 2

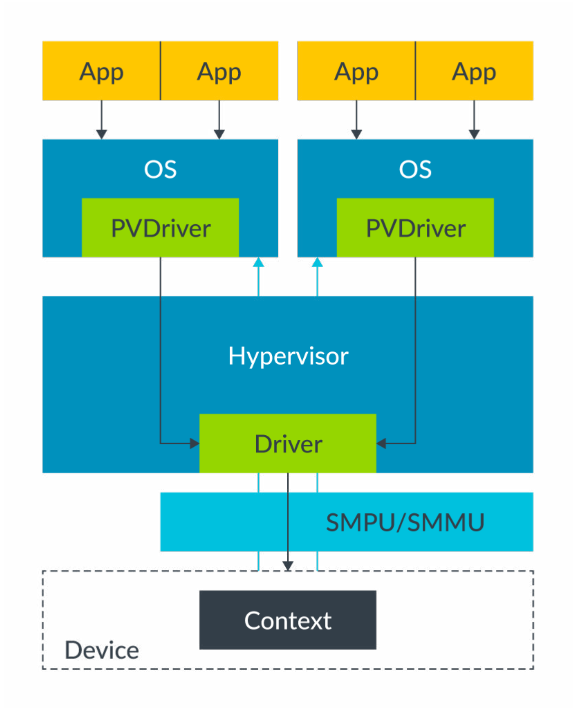
Device Emulation example



Device Paravirtualization

In this design, the VM device driver (paravirtualized driver) is aware that the device is virtualized and collaborates with the hypervisor driver to avoid trap and emulate cost, thereby increasing device virtualization performance. However, there is still a VM exit cost for calling this central driver, as well as a certification cost, since a common design principle requires that the hypervisor be as small as possible. For example USB is sometimes virtualized this way.

FIG. 3
Paravirtualization example

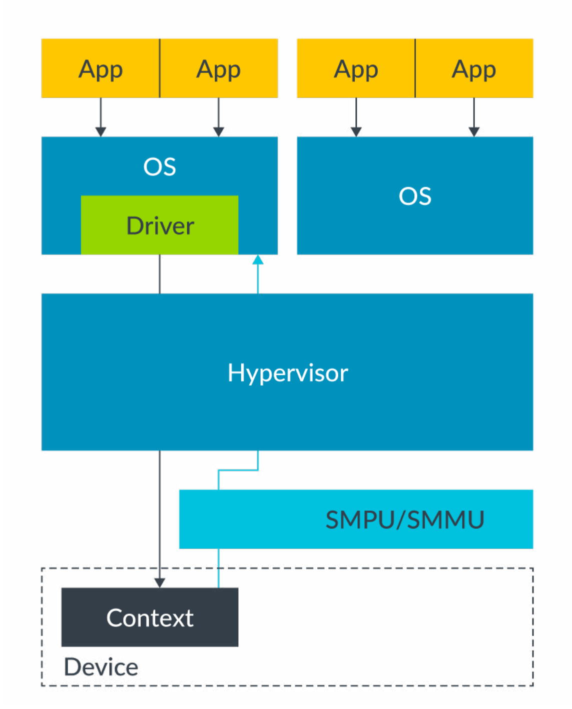


Device Pass-through

In this design, the device has a single user VM, which completely owns the device, and can read/write the device registers, share memory, and handle the interrupts. The VM driver in this case is the native driver, with the VM able to achieve near-native performance. The drawback in this case is that the device is not shared. On virtualized ECUs, LIN and CAN buses for example could be passed through.

FIG. 4

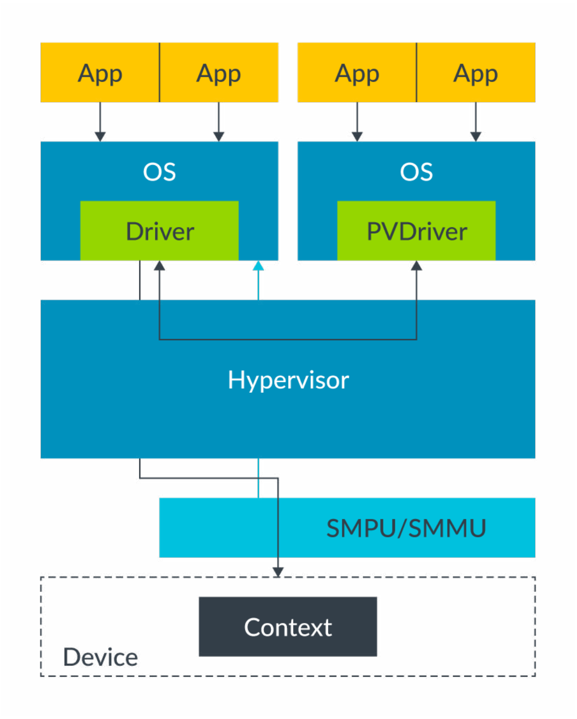
Pass-through example



Driver VM

This design is a variation of the device emulation pattern above. It mixes the pass-through approach where the device is completely assigned to a VM (the driver VM here) with the paravirtualization approach where the sharing is achieved using some device specific or more generic means (like virtio) of communication mechanism between this driver VM and the user VMs. There are typically two kinds of drivers involved in the processing in the driver VM: a backend driver receiving commands from the paravirtualized driver, and a device specific driver actually owning the hardware. There is still a performance cost, but one benefit over the device emulation design here is avoiding drivers in the hypervisor for a smaller TCB and improved security. From the guest perspective, it is paravirtualization: for example paravirtualized network drivers will be used if Ethernet is virtualized using such architecture.

FIG. 5
Device sharing with a driver VM example

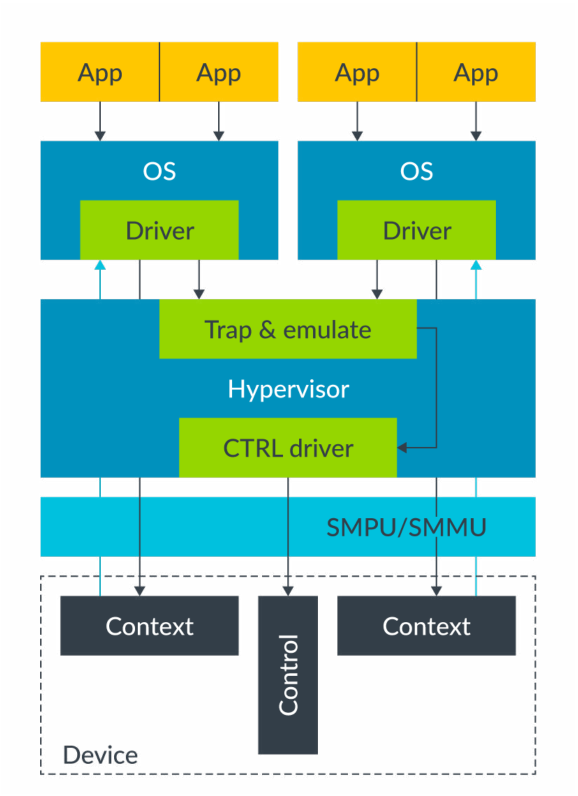


Mediated Pass-through

Mediated pass-through devices support hardware-assisted virtualization, where each device context can be independently assigned to a different VM. This approach is similar to pass-through devices, but the device control remains under hypervisor supervision and may be trapped and emulated in the same ways as regular device virtualization. Assuming device control and configuration is infrequent, this approach can also achieve near native performance. For example the Mali G78-AE GPU, with its hardware virtualization support, enables such software architecture.

FIG. 6

Mediated pass-through example

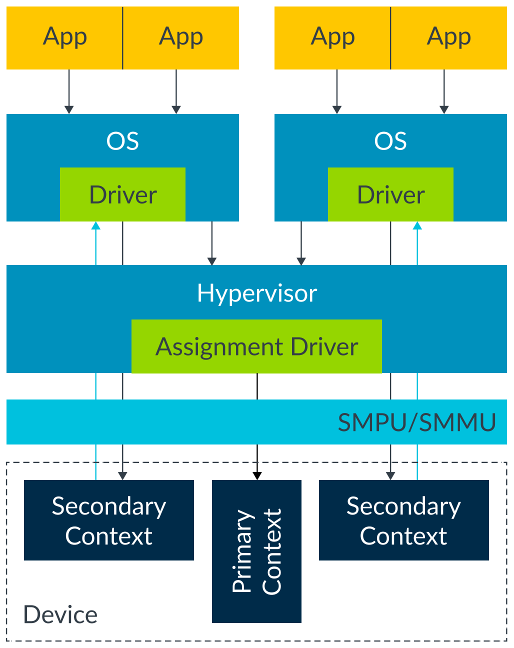


Standard Multi-context Device Pass-through

With this design, a device supporting standard hardware-assisted virtualization could be easily assigned to different VMs, with the VM wdriver collaborating with the hypervisor standard assignment driver to assign and release device contexts. Since this assignment control driver can be common to all devices, the hypervisor does not need to incorporate a device specific driver for each different device supporting this approach. For example the Mali G78-AE GPU, with its hardware virtualization support, enables such software architecture.

FIG. 7

Hardware device virtualization passthrough example



03 Device Sharing Examples

Zonal Architecture

As the whole automotive industry is shifting towards the software-defined vehicle (SDV) approach, with a central compute system and dedicated zonal controllers, virtualization is becoming a critical requirement to unlock its promises, which include more independent and manageable functions, faster testing, validations, and updates. However, proper virtualization of accelerators and peripherals may prove challenging: the following sections discuss some key architectural requirements and recipes to enable proper device virtualization on automotive platforms.

The shift of automotive architectures toward a zonal architecture to help solve scalability issues and reduce the needed cabling length, weight, and cost, requires a new generation of real-time compute which must support integration of mixed criticality software artifacts. Those mixed criticality systems (MCS) require even stronger isolation, including temporal, spatial, and fault isolation, to prevent interference while maintaining real-time support.

Such isolation may lead to a huge resource waste and poor performance if not properly handled. As a result, proper resource sharing and isolation is increasingly essential, and while virtualization seems a common choice for CPU resource sharing while maintaining sufficient temporal, spatial, and fault isolation, devices (including hardware accelerators such as GPUs, NPUs, and ISPs) and I/O peripherals also have to be shared while maintaining proper isolation.

For example, when an I/O communication peripheral is shared, it is critical that both proper isolation and a sufficient level of performance are maintained, which in some cases means including hardware support for virtualization, such as transactions filtering and separation between the PE and the device, or between the device and the memory.

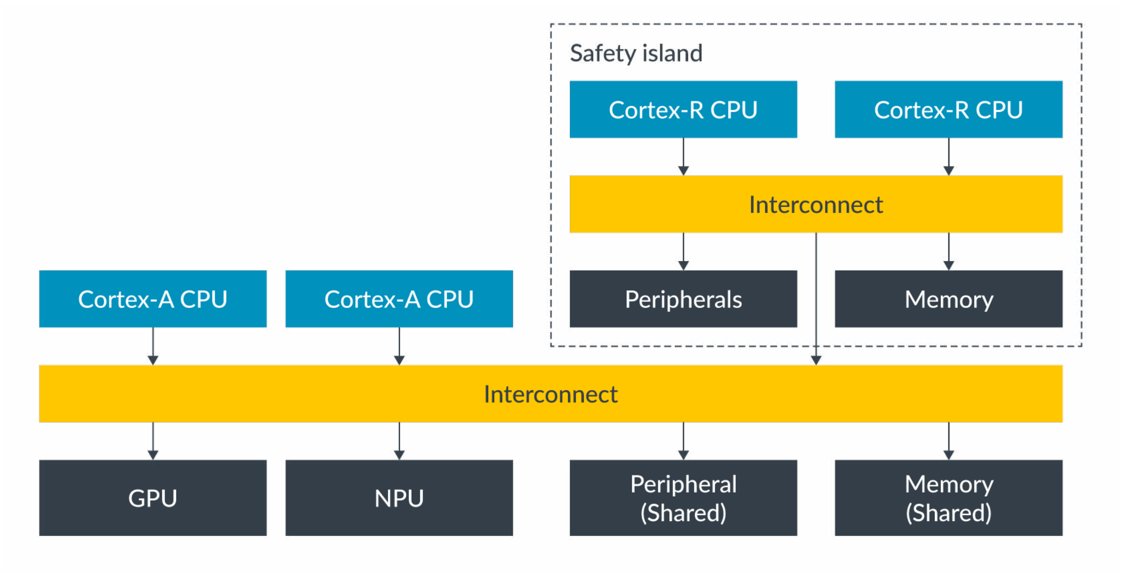
Safety Island

In a similar way, automotive SoCs increasingly rely on a safety island to manage and monitor all of the safety aspects, to enable recovery of complex issues and signal all kind of failures to external systems.

Inside such subsystem, different criticality levels must be isolated, while sharing I/O peripherals and some devices between the different clusters of CPU cores, as pictured in this figure.

FIG. 8

Safety island example



It may be impossible to address those safety island requirement when accessing shared devices while relying solely on paravirtualization. In such cases, device virtualization will require some hardware support and should follow some basic principles exposed in this whitepaper.

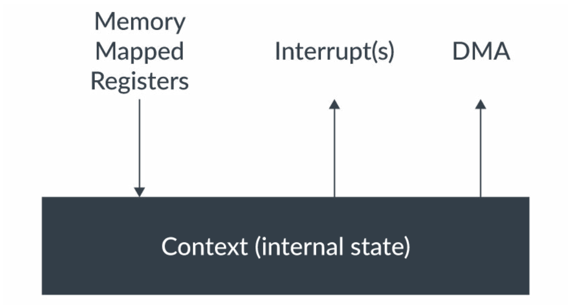
04 Device Architecture Recommendations

For a device to be virtualizable and shared among isolated software entities and for it to address use cases such as those described above, it must comply with a certain number of requirements. This section is a set of guidelines that should be considered by SoC architects to comply with those requirements. To illustrate this, we call any device complying with those guidelines a 'well-behaved device'.

Device Contexts and Assignment

This paper uses the term context to describe a portion of device functionality that can be independently assigned to software. Such context may include a state that is associated with its operation. Any embedded device (for example, hardware accelerators or peripherals that support hardware virtualization or not) has one or several such contexts.

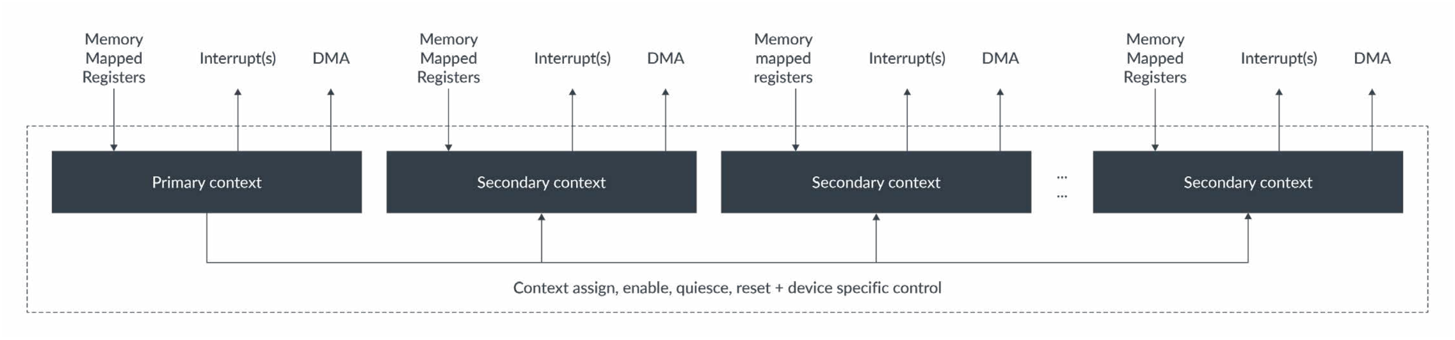
FIG. 9
Device context



By looking at all existing devices, we can observe that they all have a single primary context and zero or more secondary context(s). The primary context contains the control to manage the assignment of secondary contexts (if present) and may contain other behaviors that are usually intended to be used exclusively by the most privileged software accessing the device. All secondary contexts on well behaved devices must be independent and isolated from each other (such devices are often referred to as supporting hardware virtualization).

FIG. 10

Device with secondary contexts



We also must define what device assignment means: in this paper, assigning a context of a device to software requires that:

– The software to which the device has been assigned can directly access any memory mapped registers that are presented by the context (if any).
– Memory accesses by the device context on behalf of the software must be policed by memory protection hardware (like a System Memory Management Unit or a System Memory Protection Unit) managed by the higher privilege software responsible for providing context isolation (permitted combinations are detailed below).
– Interrupts triggered by the device context are routed to the software to which the context has been assigned.

On well-behaved devices, primary context behaviors may usually have effects that are visible from within secondary contexts. Examples of such behaviors may include:

– Device behavior that has an effect on other contexts and must be arbitrated by privileged software.
– Device behavior that is not performance critical and is accessed by lower privilege software through device emulation or paravirtualization.

While a common design pattern is to have identical secondary contexts, all secondary contexts do not have to be identical for the device to be well behaved. The secondary contexts operations must not have any effects that are visible from other secondary contexts but are permitted to have effects that are visible from the primary context to maintain isolation between software entities which have been assigned to different secondary contexts. Secondary contexts may also share device resources when performance isolation is not a use-case requirement. In that case, Arm recommends that the primary context provide mechanisms to statically or dynamically arbitrate those resources allocation to ensure that real-time constraints are met. Whether devices are shared between software entities running on Armv8-R PEs or running on heterogeneous environments, such as a mix of Armv8-R and Armv8-A PEs, they must comply with some architectural requirements to avoid compromising the system real-time and isolation properties. The following sections will expose important guidelines and rules to meet those requirements.

Reset

A first set of requirements come from device reset support, which is critical for proper real-time system operation, such as restarting a crashed VM, recovering from hardware errors or re-assigning a device context without leaking secrets. A well-behaved device must provide a mechanism to reset the primary context and confirm that the reset was completed.

The resetting of the primary context must also reset all secondary contexts so that the device can fully recover, for example, from a critical error condition, and restart from a known state upon a privileged software request. Similarly, privileged software must have the means to reset a secondary context and then confirm an individual secondary context has been reset.

Higher privilege software must have access to a reset operation to individually and independently reset each context, no matter what state the device is in. Upon reset, each context must return its internal state to a consistent known initial state.

User alterable values in this context must be returned to their reset values, and subsequent behavior must be independent from the internal state before reset to avoid leaking information and breaking the isolation between virtualized or isolated software entities.

When the context is Direct Memory Access (DMA) capable, we must avoid having the device lose track of pending transactions and receive unexpected completions messages, so a well-behaved device's contexts must drain and gracefully terminate all outstanding memory transactions upon reset. The device must not assert reset completion until this is done.

Memory-mapped Registers

When such well-behaved devices have memory mapped registers, it is also important to enable proper isolation for access to those registers, otherwise a software entity could affect a context that was not assigned to it. As such, memory-mapped registers associated with a context (primary or secondary) that is individually assignable must be in a different protection region (64B minimum, see references[1][2]) of the physical address map from the one containing the registers for every other context in the system.

In some systems, where processing elements (PEs) support Virtual Memory System Architecture (VMSA), memory-mapped registers associated with a context (primary or secondary) must be in different pages of the physical address map. Arm recommends that memory-mapped registers associated with a context expect a 64kB isolation granule, which makes it compatible with both 4kB and 16kB.

DMA

For DMA-capable contexts, a context could be used to access memory outside the allowed address spaces by the software entity to which it is assigned. One mechanism that can be used to implement this is for all memory accesses made by the well-behaved device context to be associated with a set of runtime-immutable identifiers that are unique to the context in the device. This set of identifiers must be discoverable by privileged software and used to prevent unpermitted accesses by the software entity to which the context has been assigned. For an assigned context, privileged software must be able to link the identifier to the Virtual Memory Identifier (VMID) of the software entity to which the context has been assigned.

Interrupts

Device interrupts are also of interest since they are such a critical part of device interaction, especially on real-time systems. Well-behaved device interrupts must have a pre-defined semantic (either level or edge) that do not change for the duration of the assignment to a guest. Also, isolation guarantees require that such devices must not share interrupts with any other device, and secondary contexts must not share interrupts with other secondary contexts. If interrupts associated with a primary context can originate from a secondary context, the primary context must have a mechanism to record and report which secondary context originated the interrupt. In any case, when appropriate, the privileged software agent should handle the interrupt, and inject it into the proper VM as virtual IRQ or virtual FIQ.

Errors

Errors also need to be considered when looking at device virtualization. There is a strong incentive for containing the error blast radius to avoid breaking the system isolation properties. Thus, for well-behaved devices, secondary context errors must be contained to the context that generated the error.

Memory Protection

At the system level, for well behaved devices, memory protection mechanisms are required.

For example, higher privilege software responsible for providing context isolation must ensure that the memory-mapped registers associated with an active context are only accessible through transactions initiated by the software that owns the context. This may be enforced by memory protection mechanisms in the PE or in the memory system.

All direct memory accesses from a well-behaved device must go through system memory protection mechanisms. The system configuration must be one of:

- System MMU at stage 1
- System MPU at stage 1
- System MPU at stages 1 and 2
- System MMU at stage 1 and MPU at stage 2

To facilitate a clean privilege separation model in the system, those memory protection mechanisms should be separate from those for the PE in charge of the device assignment. Privileged software may use the MMU or MPU to police accesses made from each context (note that MPU stages could be shared with PE). Sometimes, to address specific situations, one or several of the MMU or MPU checks could be disabled for a given combination of transaction attributes. Those system MPU and system MMU must be programmable at runtime (in order to support dynamic re-assignment, where a context can be re-assigned at runtime to another software entity).

System MPU and system MMU programming mechanisms (MMIO or memory) must be in different protection regions (64B minimum as per references[3][4]) of the physical address map from the regions containing the registers for every other secondary context in the system. In systems where PEs have support for VMSA, all system MPU and system MMU programming mechanisms (MMIO or memory) must reside in different pages (from each other) of the physical address map, where page size must be at least 4KB.

When being configured, all the memory protection mechanisms in the system must not allow a transaction to pass, that would not pass under the old or the new configuration setting. They must provide a way for privileged software to determine which permissions are currently active. This is key to enabling more secure and generic software for programming the system MPU.

When enabled, a system MPU or MMU must report a fault to the privileged software responsible for providing context isolation through an interrupt whenever an error condition becomes active (for example, on a first configuration operation or permission check failure).

In order to deal with interrupts in such scenarios, and enable interrupt virtualization, the system must include an Arm Generic Interrupt Controller (GIC) version 3 or newer[5].

State Diagram

A well-behaved device context must also be able to transition into some states with the properties below to enable a safe and secure device context assignment at runtime.

Those context state transitions are described in the diagram below:
When a secondary context is in the following state:

- Unassigned: The context must not try to access memory or trigger interrupts. It is expected that protection regions are setup so that lower privilege software cannot access a context which is in this state.

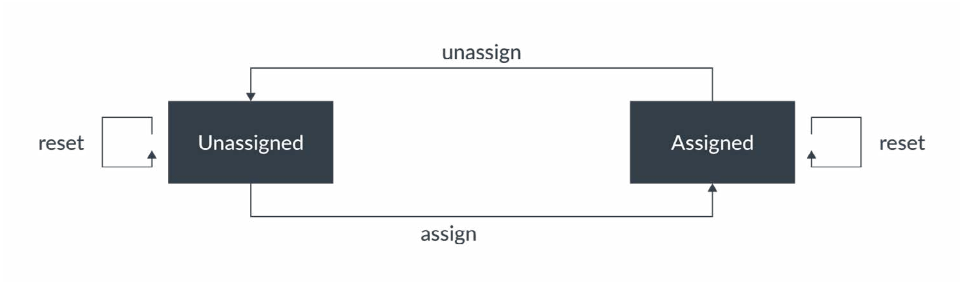


FIG. 11
Contexts state diagram

- Assigned: The context can access memory or trigger interrupts. It is expected that protection regions are setup so that a given lower privilege software can access a context which is in this state.

05 Conclusion

This whitepaper exposed a few principles for device sharing and virtualization in real-time safe systems. But this is just the beginning. As new use models, including microservices that require highly independent software components are being investigated, standardization is the only way to reap the benefits of SDVs in the automotive industry, and must happen in the near future to lower the costs and enable more robust approaches. For example, cost-effective scaling means having a single and more flexible platform, rather than several less flexible ones, something that cannot be achieved without proper standardization.

06 Glossary

Term	Meaning
E/E	Electrical/Electronic
FIQ	Fast Interrupt reQuest
IRQ	Interrupt ReQuest
MCS	Mixed Criticality System
MCU	Microcontroller Unit
MMIO	Memory-Mapped I/O
PE	Processing Element
SDV	Software Defined Vehicle
SOAFEE	Scalable Open Architecture for Embedded Edge
System MMU or SMMU	System Memory Management Unit
System MPU or SMPU	System Memory Protection Unit
TCB	Trusted Computing Base
VM	Virtual Machine
VMSA	Virtual Memory System Architecture

07 References

[01] Arm Architecture Reference Manual Supplement Armv8

for Armv8-R AArch64 architecture profile. (ARM DDI 600) Arm Ltd.

[02] Arm Architecture Reference Manual Supplement Armv8

for Armv8-R AArch32 architecture profile. (ARM DDI 568) Arm Ltd.

[03] Arm Architecture Reference Manual Supplement Armv8

for Armv8-R AArch64 architecture profile. (ARM DDI 600) Arm Ltd.

[04] Arm Architecture Reference Manual Supplement Armv8

for Armv8-R AArch32 architecture profile. (ARM DDI 568) Arm Ltd.

[05] Arm® Generic Interrupt Controller Architecture Specification

GIC architecture version 3.0 and version 4.0. (ARM IHI 0069)
ARM Ltd.