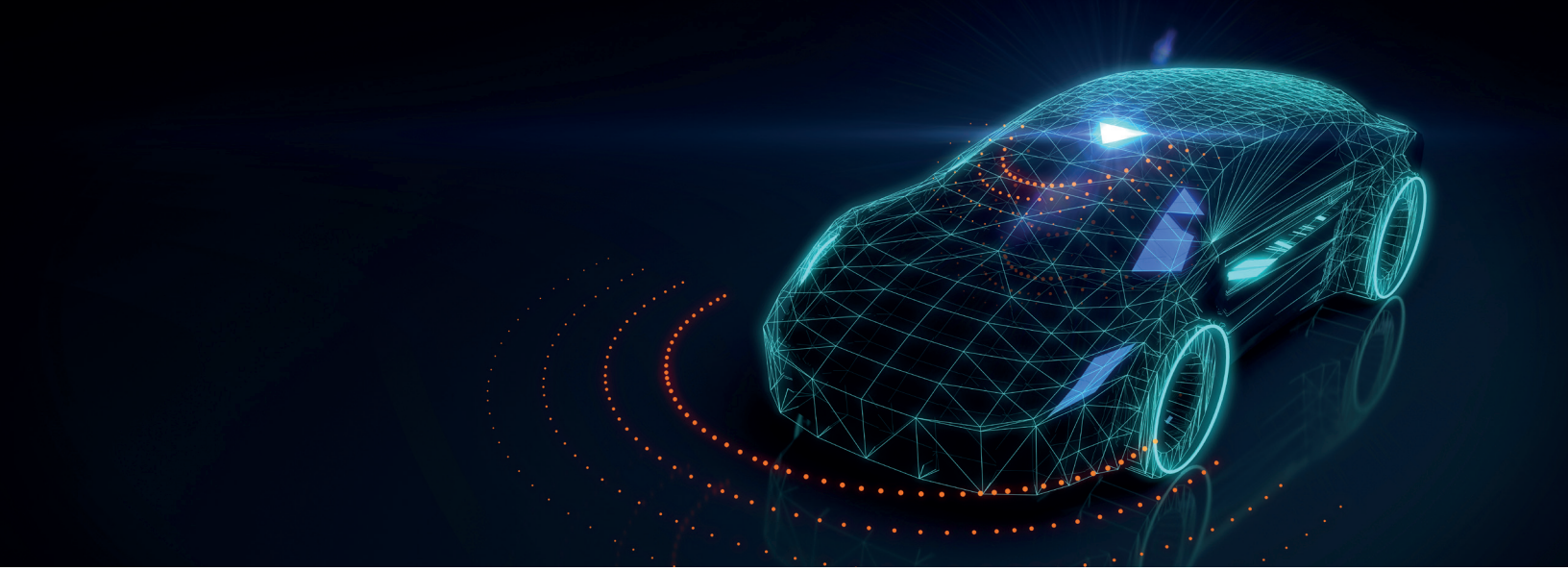
WHITE PAPER

Best Practices for Armv8-R Cortex-R52+ Software Consolidation

Dr Paul Austin, Principal Software Engineer, ETAS
Dr Andrew Coombes, Senior Product Manager, ETAS
Paul Hughes, Lead System Architect and Distinguished Engineer ATG, Arm
James Scobie, Director Automotive Product Management, Arm
Bernhard Rill, Director Automotive Partnerships EMEA, Arm

Contents



Introduction

Vehicle electrical/electronic (E/E) architectures are evolving towards the centralization of compute resources. This initially happened in domain controllers before moving to zonal and centralized approaches.

As multiple real-time functions are consolidated into zonal controllers, the requirement for the processor performance increases, as does the sophistication of the operating system and software. The industry is increasingly turning to Armv8-R based solutions like the Cortex-R52 and Cortex-R52+ CPUs (summarized in the paper as Cortex-R52+) to enable this software integration vision. Several automotive chip manufacturers have already incorporated these processors into their designs for high-performance microcontrollers for zonal platforms and safety islands. Meanwhile, automotive software providers have established solutions that integrate with Armv8-R, the Arm architecture used in this processor family. A state-of-the-art overview of this industry trend is summarized in this Arm Blueprint article.

The new system hardware and software must simultaneously meet the requirements of every individual workload hosted on a device. These include:

- Satisfying the software dependencies of a workload, including libraries, operating system calls (including access to Input/Output), and Application Binary Interfaces (ABI). Specific versions of these may be required. Where a single operating system is unable to simultaneously satisfy all dependencies, system software may include more than one operating system.
- Performance, including the determinism required for real-time workloads may have different hard real-time response time requirements, ranging from a few microseconds upwards. Failure to meet hard real-time requirements results in an incorrect operation for the workload. Some workloads may have leaner real-time requirements where failure to meet these results in degraded performance. Other workloads have no dedicated real-time requirements, so these software artefacts are executed on a best effort basis.
- For functionally safety workloads, satisfying the workload's assumptions of a correct execution environment along with the provision of any assumed external safety mechanisms is fundamental. The Automotive Safety Integrity Level (ASIL) of the execution environment and safety mechanisms must be as high or higher than that assigned to the workload.
- For workloads with less safety relevance, it is highly desirable that the ASIL of the workload is not increased simply because other higher ASIL workloads are present on the device.
- Meeting security requirements, such as confidentiality, integrity, privacy, and authenticity, through ensuring sensitive data is not accessible to other workloads.
- The ability to update individual workloads, including firmware over-the-air (FOTA). This also encompasses a range of other topics, such as authentication of workloads, secure boot and system-level updates. FOTA for hypervisors is a large topic, which cannot also be addressed in this white paper.

- For workloads derived from pre-existing (legacy) applications, a desirable option is to integrate the workload with minimal adaptation. When integrating workloads that were designed for standalone system hardware, software must protect against any behaviours with side effects that affect other applications in the system.
- For workloads which relate to regulated applications, it may be necessary to obtain certification (for example, in the case of On-Board Diagnostics (OBD)-relevant applications). To avoid the need for re-certification every time another workload changes, it is important to demonstrate that the other workloads do not interfere with the certified workload.

By hosting multiple workloads, system hardware and software must provide appropriate isolation between workloads to ensure that one workload cannot cause another workload to fail to meet its requirements. Where multiple operating systems are required, similar isolation requirements exist between each operating system.

For functional safety, this type of isolation is known as freedom-from-interference (FFI), and requires mechanisms to ensure faults related to one workload do not cause failure of the execution environment and system safety mechanisms provided to another workload.

A system that provides this level of isolation between workloads also brings the advantage of allowing each workload to be developed (and debugged) in isolation from other workloads. This is especially important if the workloads are coming from different suppliers.

System hardware and software provide the following isolation mechanisms that are used to meet these requirements:

- Logical isolation. Isolation of state belonging to different workloads using a privilege model and memory protection mechanisms.
- Timing isolation. Scheduling of private and shared resources, partitioning and monitoring of shared resources, and managing watchdog timers to detect timing violations.

Cortex-R Outline

Arm has a portfolio of CPU processors that are designed to address a wide range of computing, from the smallest, lowest power microcontrollers to ultra-high performance server class computing. The Cortex-R processors have been developed to enable applications where there are demands for real-time processing and are applicable to a range of different uses cases, not least of all in automotive applications where systems must respond in short and deterministic timeframes to successfully meet the requirements of the system deadlines. In many cases, these applications also include functional safety (and security) requirements that add to the challenges faced by system integrators and developers. Cortex-R processors, like the Cortex-R52+ can be used in standalone microcontrollers (MCUs) or as additional cores in a SoC (System on Chip) design, for example as a safety island.

The first Cortex-R processors, such as Cortex-R5, were built on the Armv7-R architecture. However, since then the architecture has evolved, with Arm's Cortex-R52 and Cortex-R52+ processors implementing the Armv8-R architecture which helps address the increasing complexity of automotive real-time software and the transition from discrete dedicated controllers to those where functions are centralized and combined. The Armv8-R architecture adds support which enables the better control of software within a single processor, providing isolation of code and enabling reproducible and understandable behaviour including virtualization in a real-time processor.

The Cortex-R52 and Cortex-R52+ processors are highly configurable and can be defined to suit the implementors application requirements. Some of the configurability is described in Table 1.

TABLE 1
CortexR52(+) Example, Config Parameters

Config Parameter	Cortex-R52/Cortex-R52+
Cores per cluster	Configurable 1-4
Stage 1 MPU	8,16,20,24 Regions
Stage 2 MPU	0,8,16,20,24 Regions
Logic fault detection	Dual Core Lockstep, Split/lock
Interface Protection	Optional
Interconnect Protection	Optional
Floating Point	Single/Double precision
SIMD	NEON
GIC Interrupts	Configurable 32-960 SPIs

As part of the Armv8-R architecture, these processors provide an additional exception level to those of the user space, Exception Level 0 (EL0), and Operating System space, Exception Level 1 (EL1). This new Exception Level 2 (EL2) can be used to help the management of software on the processor with the aid of a hypervisor/separation kernel, which simplifies how partners control software access to shared resources and its interaction. This can be used to maintain isolation between tasks on a single processor running on the same operating system, or across multiple operating systems.

Together with the new Exception Level comes the addition of a two stage Memory Protection Unit (MPU), which is able to enforce the access performed by the processor to different resources. The Operating System is able to control the MPU for its resources at EL1, but the processors can be implemented to add this additional second stage of the MPU, which is only configurable from the EL2 from where the hypervisor can run.

Access to resources is managed with software running at the new higher Exception Level 2. Application tasks can request access to the required resources through this software, which enforces the access with the two level Memory Protection Units (MPU). This approach is not limited to two different criticality levels, but can support many different contexts with differing protections. Unlike a Memory Management Unit (MMU) the availability of an MPU can offer the access management from the Cortex-R processors to the system resources without the introduction of additional, potentially schedule breaking, delays to search and load page tables from memory. These are hard to manage, as well as being difficult to evaluate and guarantee their timely completion.

The two levels of MPU are:

- The EL1 MPU, which is managed by the operating system to enforce the separation of the Operating System from application tasks/ISRs and also the separation of application tasks/ISRs from each other. The EL1 MPU can be programmed by code running at EL2 or EL1.
- The EL2 MPU, which can only be programmed by code running at EL2 and is used by a hypervisor to provide additional separation.

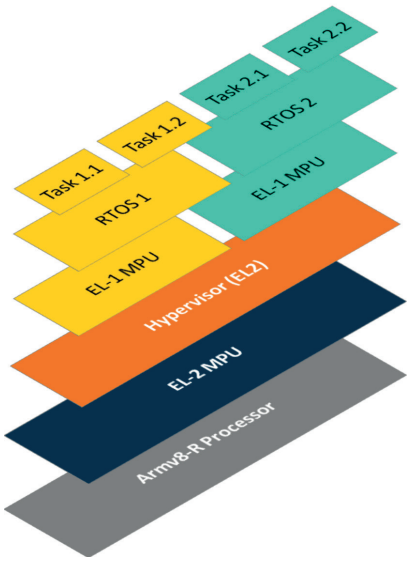


FIG. 1
Armv8-R Exception Levels

The Cortex-R52+ provides information outside the core to enable the system to establish and maintain control of accesses based on the running software. This is achieved by propagating the virtual Machine ID (VMID) for device transactions to enable the system to manage access to those resources. In the case of Cortex-R52+, this is further extended by supporting buffers and memory transaction and requests, which are made directly from a hypervisor at EL2.

These Cortex-R processors integrate their own Generic Interrupt Controller (GIC) shared by all CPUs within the cluster to deliver low latency interrupts from the system. This can flexibly assign and prioritise Shared Peripheral Interrupts (SPI) to any of the cores in the cluster. The GIC supports the ability to signal both physical and virtual interrupts and can trap interrupt accesses to EL2 to virtualize interrupts.

The processors have Tightly Coupled Memories for highly deterministic, low latency access to code and data by the cores. They have multiple interfaces to external resources, including SRAM, main memory and devices. Resources accessed by an interface are assigned based on their address location with implementors able to flexibly allocate the space in the memory map they use and manage the allocation of resources to be privately assigned to a virtual machine.

Software Integration Mechanisms

As the amount of software in a vehicle increases, progressively more-and-more applications are being integrated onto one microcontroller. This can be seen particularly in domain/zonal controllers that provide the bridge between the very powerful central vehicle computers (typically using Arm Cortex-A cores and running a POSIX based operating system and Adaptive AUTOSAR), and the simpler ECUs on the mechatronic rim, which typically use Arm Cortex-R and Cortex-M cores.

Hypervisors and Virtual Machines

Microcontrollers using the Cortex-R52+ can support systems that integrate applications with the necessary separation. Each application is run inside its own separated instance – usually referred to as a *partition* or *virtual machine (VM)*.

A VM is typically composed of:

- Some physical or virtual processor cores
- Some memory
- Some physical or virtual peripherals
- Some physical or virtual configuration registers

The software that manages VMs is usually called a *hypervisor*, *separation kernel* or *VM manager*, which, on a Cortex-R processor, runs at Exception Level 2 (EL2) privileged level.

To a greater or lesser extent, a hypervisor creates the illusion to the *guest software* running inside a VM that it is running on its own microcontroller and not sharing the microcontroller devices with other guest software in other VMs.

Note that the Armv8-R Cortex-R processors (and similar devices) do not provide Memory Management Units (MMUs). A hypervisor running on a Cortex-A device can use its MMU to present each VM with a completely separate virtual address space. For example, the guest software running inside each VM can be linked to run at the same address and use the same range of memory addresses for data. The Memory Protection Unit (MPU) provided by the Cortex-R52+ allows a hypervisor to protect one VM's memory from another VM but does not allow each VM to have a separate virtual address space.

One physical processor core can host multiple *virtual cores* by context switching between the virtual cores in the same way that operating systems context switch between processes. A virtual core's context is the values of the general-purpose registers, floating-point registers, some system configuration registers and the configuration of the EL1 MPU.

Where legacy software is being run inside a VM, we want the VM to look as much as possible like a real microcontroller to avoid the need to change the legacy software other than by re-linking so that the guest software running in each VM uses separate memory.

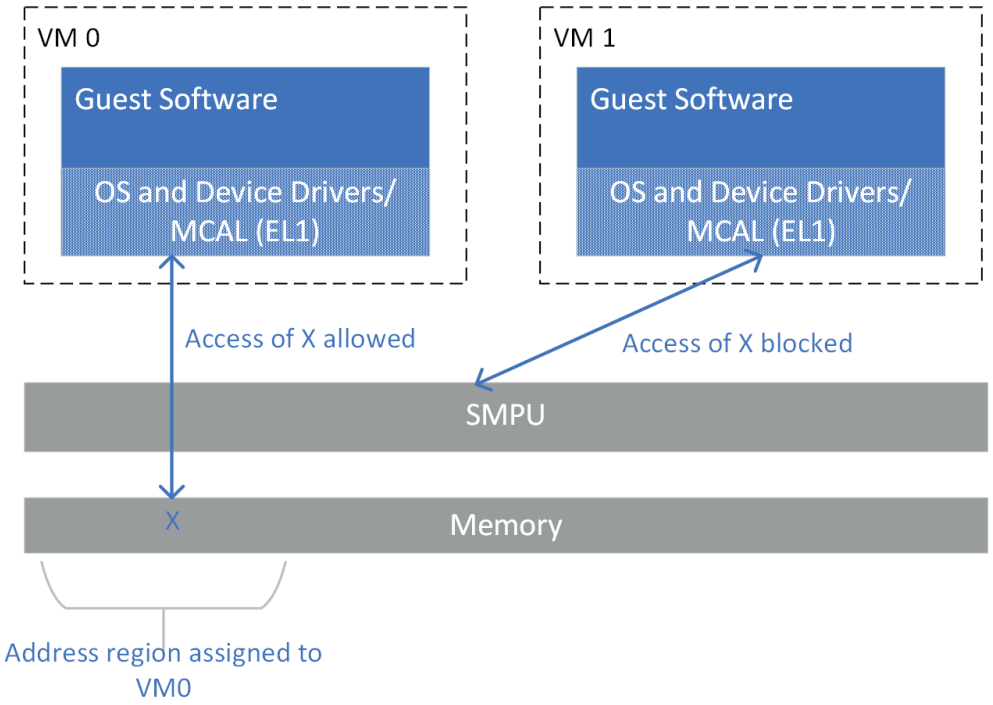
Using SMPUs and Peripheral Protection Mechanisms

Microcontrollers that contain Cortex-Rs will normally include a *system-level memory protection unit (SMPU)*. The primary role of an SMPU is to control which bus managers (e.g DMA controller) can access which memory addresses. Cortex-R processor cores and other microcontroller components, such as Cortex-M cores and some peripherals, can be bus managers. Typically, an SMPU will have a collection of regions. Each region has a configurable start address, a size and is assigned to one or more bus manager (or in more advanced designs, to one or more VMs using a VM identifier stored in the Cortex-R52+'s VSCTLR.VMID register). A bus manager (or VM) can only access memory in regions assigned to it.

Microcontrollers may also include *peripheral protection* mechanisms to allow peripherals to be assigned to bus managers (or VMs). Here a peripheral is assigned to one or more bus manager (or VMs) and the peripheral protection mechanism then prohibits any other bus manager (or VM) from accessing the peripheral's registers.

By using SMPUs and peripheral protection mechanisms we can achieve cluster-level separation. That is, a microcontroller's memory and peripherals, can be partitioned amongst multiple VMs where a VM contains all of the cores in a Cortex-R core cluster. Relying solely on the above mechanisms would not allow us to have multiple VMs in the same cluster if the VMs have different safely levels (e.g., different ISO 26262 ASIL levels). Each cluster has a *Generic Interrupt Controller (GIC)* that is used to route interrupts to the cores in the cluster. Each core has a separate GIC redistributor to handle Software Generated Interrupts (SGI) and Private Peripheral Interrupts (PPI), but the GIC distributor used to handle SPI interrupts is common to all cores in the cluster. If we allowed multiple VMs in the same Cortex-R52 core cluster to write to the memory mapped GIC distributor registers, a VM could interfere with another VM by (accidentally or maliciously) changing the other VM's interrupt configuration.

FIG. 2
Partitioning Memory with an SMPU



An advantage of using SMPUs, rather than just core MPUs, is that it allows us to create VMs that include not just Cortex-R52+ cores but also other DMA capable components that may be in the microcontroller and are connected to the same memory bus. For example, microcontrollers may include clusters of Cortex-R52+ cores and some special purpose Cortex-M cores.

Using EL2 for Para-virtualization

In addition to protection mechanisms like SMPUs and peripheral protection provided by the microcontroller, the Cortex-R52+ itself includes features to support virtualization. One of these features is the EL2 privilege level. EL2 is more privileged than the EL1 (supervisor) level used by an operating system and the EL0 (user) level used by application code. A hypervisor runs at EL2 and code inside a VM, the guest software, runs at EL1 or EL0.

The HVC (hypervisor call) instruction can be used by code running at EL1 to make a request to a hypervisor in the same way that the SVC (supervisor call) instruction can be used by application software to make a request to an operating system. When software running at EL1 executes a HVC instruction, the Cortex-R52+ core switches to EL2 and takes a *Hyp-mode* entry exception. The hypervisor handles this exception and then returns to the guest software at EL1.

The HVC instruction allows for *para-virtualization*. This is where guest software is aware that it is running in a VM, and the hypervisor provides an API (using HVC instructions) that the guest software uses to make requests to the hypervisor, to device drivers plugged in to the hypervisor (*EL2 device drivers*), or to device drivers running in other VMs.

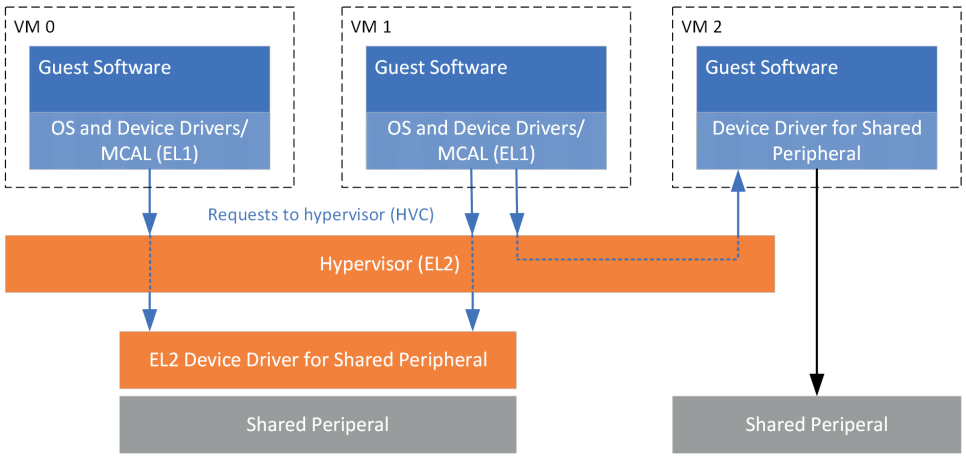


FIG. 3
Para-virtualization

As an example, consider how para-virtualization can be used to allow multiple VMs to exist inside the same cluster despite the shared GIC distributer. The SMPU (or the core MPU) is configured so that the VMs do not have access to the GIC's memory-mapped registers. When guest software wants to change its interrupt configuration, it makes an API request to the hypervisor. The hypervisor carries out the necessary GIC configuration having first checked that the requested changes will not interfere with another VM.

Para-virtualization can also be used to allow peripheral sharing and creation of virtual peripherals. A peripheral, such as an Ethernet controller, can be shared in much the same way as the GIC. Completely virtual peripherals can also be created. For example, one might create a virtual Ethernet controller used for communication between VMs running on the same microcontroller. In both cases, the hypervisor would contain an EL2 device driver that either managed access to the shared peripheral or implemented the virtual peripheral. This is analogous to the way that an Operating System uses devices drivers to manage access to peripherals shared by multiple processes or tasks.

Para-virtualization can be used as a solution for peripherals that do not, or do not fully, support virtualization in hardware. Ideally peripherals would support virtualization as described in "Device virtualization principles for real time systems" to avoid the need for para-virtualization – at least on the data plane. Para-virtualization (and trap-and-emulate) will always add some additional cycles when compared to device pass-through (where a peripheral is driven directly by guest software without hypervisor intervention), but unless a peripheral supports virtualization, para-virtualization (or trap-and-emulate) is likely to be needed to allow the peripheral to be shared between VMs. In the case where para-virtualized code is only executed infrequently, the overhead of para-virtualization may be acceptably small e.g., where a control plane must be para-virtualized but a data plane does not.

Using EL2 for Trap-and-emulate

In some cases, para-virtualizing guest software may not be possible. In these cases, *trap-and-emulate* can be used.

When code running at EL1 or EL0 makes a memory access prohibited by the EL2 MPU, the Cortex-R52+ processor switches to EL2 and takes a Hyp-mode entry exception. This feature can be used by a hypervisor to allow emulated access to peripherals with memory mapped registers. The EL2 MPU is configured to prohibit access to the registers. When guest software reads or writes a register, a Hyp-mode entry exception occurs at EL2. The hypervisor works out which register the guest software was reading or writing by examining the Cortex-R52+'s Hyp Syndrome Register (HSR) – which contains details of why an exception occurred - and Hyp Data Fault Address Register (HDFAR)– which contains the memory address being accessed when an exception occurred - and either emulates access to the register itself or delegates to an EL2 device driver.

Trap-and-emulate can be used to access the shared GIC distributer. The EL2 MPU is configured so that guest software access to GIC memory mapped registers causes an exception. When the exception occurs, the hypervisor carries out the GIC register access having first checked that the register access will not interfere with another VM.

Trap-and-emulate can provide a means for guest software to access shared and virtual peripherals managed by EL2 device drivers. When integrating legacy software, trap-and-emulate can be used to emulate a peripheral not present in the microcontroller.

Trap-and-emulate has the advantage that guest software does not need to be modified to run in a VM. However, para-virtualization is usually more performant because:

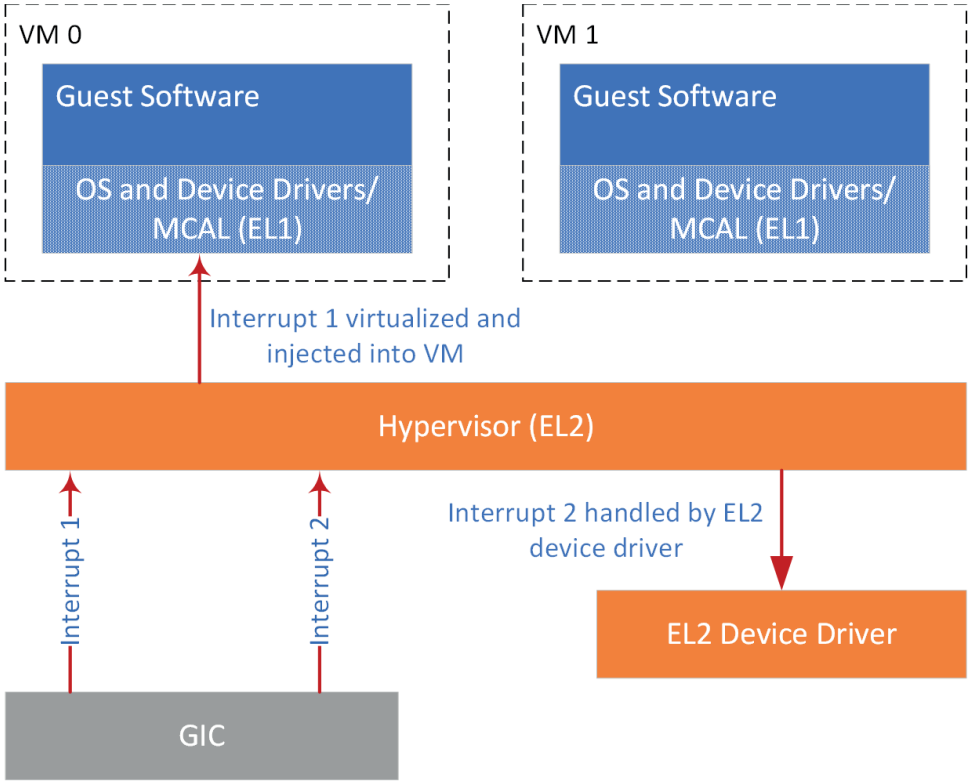
– Operations on peripherals can be requested at a higher level of abstraction, which results in few switches to the hypervisor (e.g., we might have a "configure interrupt" operation that is equivalent to multiple accesses to GIC registers).
– The hypervisor does not have to work out what the guest software was doing when the Hyp-mode entry exception occurs.

Interrupt Virtualization

EL2 alone does not allow us to share or virtualize interrupt driven peripherals. The Armv8-R architecture defines that normally when an interrupt occurs it interrupts the currently running code at the current privilege level. For example, if an IRQ occurs while code is running at EL1 then the interrupt will be taken at EL1 using the IRQ entry in the EL1 vector table, but if the IRQ occurs while code is running at EL2 then the interrupt will be taken at EL2 using the IRQ entry in the EL2 vector table.

To solve this, the Cortex-R52+ supports interrupt virtualization. When interrupt virtualization is enabled (by setting the IMO and FMO flags in the HCR register and setting the EN flag in the ICH_HCR register), an FIQ or IRQ interrupt (exception) always results in the Cortex-R52+ switching to EL2 and taking an FIQ or IRQ interrupt using the EL2 vector table. The hypervisor can then either handle the interrupt or use features of the Cortex-R52+ core to virtualize the interrupt (by using list registers), so that when guest software runs at EL1/EL0 it takes the virtual interrupt at EL1 using the EL1 vector table. To the guest software, the virtual interrupt is indistinguishable from a real interrupt. When guest software disables interrupts or changes the priority mask register value, only virtualized interrupts routed to the guest's VM are affected, not real interrupts being taken at EL2. Therefore, guest software cannot stop interrupts being taken by the hypervisor.

FIG. 4
Interrupt Virtualization



Since all interrupts are initially handled by the hypervisor, the hypervisor can decide if an interrupt should be handled by the hypervisor itself, by an EL2 device driver, or should be virtualized and injected into to a VM. This allows interrupt driven shared/virtual peripherals to be handled. EL2 device drivers can also inject virtual interrupts into VMs if needed.

Interrupt virtualization also allow "remote control" of VMs. For example, a privileged management VM or EL2 device driver running on one physical core can generate an interrupt in a second physical core to request that the hypervisor do something on the second core, such as shutting down or re-starting a VM running on the second core.

Of course, nothing comes for free, and interrupt virtualization adds to the total time taken to process an interrupt. The exact overhead depends on many factors, including the arrival pattern of interrupts and how many different GIC interrupts are being used. There are two timing concerns to be aware of related to interrupt virtualization:

1. The processing at EL2 to virtualize an interrupt consumes processor time.
2. Since prioritization of virtual interrupts is independent to prioritization of real interrupts, guest software may see timing anomalies.

   Some examples in this context: consider two interrupts A and B. A is higher priority than B. Both interrupts are routed to the same VM using interrupt virtualization.

   i. If A occurs and the EL2 and EL1 handling of A completes before B occurs, then guest software will see a small increase in latency for both interrupts due to the EL2 handling.

   ii. If both interrupts occur at the same time, the EL2 handling for both interrupts will occur before any EL1 handling, and the guest software will see a double increase in latency for A, but no increase in latency for B.

   iii. Now imagine that A occurs and reaches the EL1 handler in the guest software and then B occurs. The EL2 handling of B will pre-empt the EL1 handling of A even though A is higher priority.

To help quantify this GIC virtualization behaviour, we include the results of some experiments carried out with ETAS' RTA-HVR hypervisor. In these experiments, we compared the interrupt latency of Cortex-R52+ cores running the RTA-OS Operating System without a hypervisor to the same Cortex-R52 cores running RTA-OS as guest software inside a VM. Two cores inside the same Cortex-R cluster were used. The first core triggered interrupts in the second core by setting bits in the GIC `ISPENDR` registers (Interrupt Set Pending Registers that are used by software to trigger interrupts). The latency is the number of timer ticks between the interrupt being triggered and the start of the (fully Operating System managed – i.e., AUTOSAR Category 2) ISR. The timer was configured to run at the same speed as the processor clock. The exact value of the latency will depend on the type of memory used for code/data and the cache configuration, so in the following results it is important to focus on the comparison between the hypervisor and non-hypervisor cases rather than the absolute values.

TABLE 2

Interrupt Number	Latency (CPU Cycles) No Hypervisor	Latency (CPU Cycles) Hypervisor	Ratio Hypervisor/No Hypervisor
1	1129	2067	1.83
2	1120	2053	1.83
3	1120	2008	1.79
4	1125	2002	1.78

Table 2 shows what happens when the first core triggers four different interrupts with a large delay between interrupts so that all interrupt handing has completed on the second core before the next interrupt is triggered. In this case, we see an increase of around 80 percent in interrupt latency. In this case, the Operating System configuration does not contain any untrusted code (i.e., all application code is running at EL1).

TABLE 3

Interrupt Number	Latency (CPU Cycles) No Hypervisor	Latency (CPU Cycles) Hypervisor	Ratio Hypervisor/No Hypervisor
1	2046	2916	1.43
2	2023	2817	1.39
3	2463	3381	1.37
4	2454	3364	1.37

Table 3 contains results from the same setup as 2, except that the Operating System configuration includes untrusted tasks and ISRs. Here we see that the additional work that must be done by RTA-OS to manage untrusted code means the work done at EL2 to virtualize interrupts is a smaller proportion of overall interrupt latency.

TABLE 4

Interrupt Number	Latency (CPU Cycles) No Hypervisor	Delta Between Latency N-1 and N for No Hypervisor	Latency (CPU Cycles) Hypervisor	Delta Between Latency N-1 and N for Hypervisor
1	1255	1255	5047	5047
2	2674	1419	6366	1319
3	4089	1415	7714	1348
4	5505	1416	9051	1337

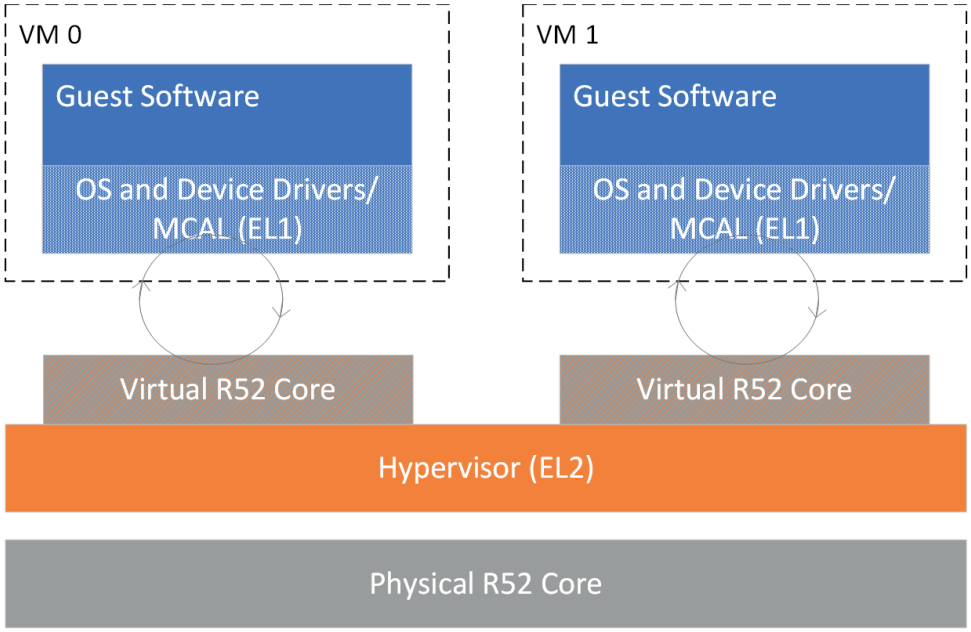
Table 4 shows what happens if the four interrupts are triggered at the same time. The lower the interrupt number, the higher its priority. For the case without a hypervisor, we see the expected behaviour given interrupt prioritization. The ISR for interrupt number 1 runs first and blocks the other interrupts until it has been completed. The ISR for interrupt number 2 then runs and blocks the other interrupts until it has been completed. And so on. The time between the ISR for interrupt N-1 starting and the ISR for interrupt number N starting is approximately the same (the ISRs executed very little code). However, with a hypervisor present we see quite different behaviour. The EL2 handling for all four interrupts occurs before interrupt number 1 is handled by its EL1 ISR. Therefore, we see a much larger interrupt latency for interrupt number 1 than for the subsequent interrupts.

Virtual Processor Cores

We can also take advantage of interrupt virtualization to support virtual processor cores. For example, a timer interrupt can be handled by the hypervisor and used to drive a virtual core scheduler that decides when to context switch between virtual cores. Since guest software cannot block interrupts being taken at EL2, broken or malicious guest software cannot deny processor time to other guest software. Interrupts that arrive for a virtual core that are not currently running can be virtualized, queued in software, and injected into the virtual core when it next runs.

Switching between virtual cores often involves re-programming the EL2 MPU. Care must also be taken with co-processor 14 and 15 configuration registers (these registers are used to configure various aspects of the processor, such as endianness, whether or not caching is enabled and whether or not the EL1 MPU is enabled). Some of these registers affect the physical core in ways that could affect all virtual cores on the physical core and the hypervisor. Other registers have effects that might be virtual core specific, such as EL1 MPU region registers. Registers in the latter category need to be part of a virtual core's context. For registers in the former category, the Cortex-R52+ can be configured to generate a Hyp-mode entry exception at EL2 when they are accessed. The hypervisor can then in some way emulate access to these registers.

FIG. 5
Virtual Cores



If multiple virtual cores are hosted by a single physical core, then consideration must be given to how virtual cores are scheduled. The simplest approach is to use a static TDMA (Time-division multiple access) algorithm. A TDMA algorithm has a very low run-time overhead, is easy to understand, and is easy to work out when a virtual core will run in wall-clock time. The disadvantage of a purely static algorithm is that it can lead to long latencies when handling asynchronous events (e.g., interrupts). It may be possible to avoid long latencies through the careful construction of the static VM schedule to ensure that an interrupt never has to wait for too long before the VM that handles it runs. However, this may require detailed understanding of interrupt worst-case execution times.

An alternative way for a system to handle asynchronous events with short latencies is to use a dynamic scheduling algorithm. An example of a dynamic scheduling algorithm is *Reservation Based Scheduling* with each virtual core being a *Deferrable Server*. Such an algorithm has been used in some versions of the ETAS RTA-HVR hypervisor. This gives much shorter latencies when handling asynchronous events at the expense of a higher run-time scheduling overhead and the need to para-virtualize the guest Operating System so that it can yield its virtual core when the operating system has no tasks to run.

Using virtual processor cores gives the system designer flexibility:

- A VM can be created that contains more cores than would be available if only physical cores were used. The extra cores might make structuring software easier – in the same way that threads are used in an operating system.
- Multiple VMs may be hosted by a single physical core.

However, context switching between virtual cores has a non-trivial overhead, as the hypervisor needs to save one virtual core's general-purpose registers, floating-point registers, relevant configuration registers and EL1 MPU settings, and then re-load these for another virtual core. A virtual core context switch is analogous to a process context switch in an Operating System.

Using virtual cores effects interrupt latency. With a static scheduling algorithm, an interrupt that arrives for a virtual core that is not currently running will not be handled until the virtual core next runs. With a dynamic scheduling algorithm that automatically switches to the virtual core that handles an interrupt, the virtual core context switch time will be added to the interrupt latency.

A system designer needs to evaluate which applications can make use of virtual processor cores. If the overhead of context switching virtual cores is not acceptable, the system designer should consider different options.

Software Integration Recommendations

Unfortunately, there is no "one size fits all" approach to integrating multiple applications into a microcontroller. The previous section has outlined several mechanisms that can be used to enable integration. Which mechanisms are appropriate will depend on the types of application being integrated.

Decide if Hypervisor-type Separation Is Needed or Would OS-level Separation Be Better?

The mechanisms described above allow a hypervisor to create the illusion that guest software has its own microcontroller and to enforce separation between the VMs running the guest software. If the applications to be integrated are tightly coupled (tightly coupled means that applications make synchronous functions calls into each other or rely on activities within all applications being scheduled by the same scheduler), then integration using Operating System containers may be more appropriate. AUTOSAR operating systems allow tasks and ISRs to be grouped into containers called "*Operating System applications*" and MPUs can be used to separate these different containers. Integration at the Operating System level means that a single scheduler is used to schedule all tasks and tightly coupled communication is better supported. New initiatives like AUTOSAR Flex-CP are being developed to support more dynamic integration of applications into Classic AUTOSAR systems.

Treat Each VM as a Separate ECU Connected by a Network

Where running each application in a VM is the best approach, it is important to treat each VM as a sperate Electronic Control Unit (ECU) connected by a network. Each VM likely contains guest software with its own Operating System and scheduling policy. Tight coupling between tasks in different VMs, both in terms of scheduling and communication, is very hard to manage and likely to result in a fragile system. It is better to treat each VM as a separate ECU and use the mechanisms built into AUTOSAR to handle systems built from multiple ECUs. This leads to a more robust, simpler, and easier to reason about system.

Use Core-local and Cluster-local Resources

It is better to use resources like memory and peripherals that are "close" to the core that uses them. Each Cortex-R core has TCMs that are much faster to access than other types of RAM. Often microcontrollers have cluster local Flash and RAM, and in some cases peripherals (e.g., CAN and LIN controllers) can be assigned to a cluster. As well as usually being faster, using cluster local resources often results in less memory bus contention because a core accessing cluster local resources may not have to contend with cores in other clusters for access to the memory bus.

Note that using resources close to a core may limit any migration of virtual processor cores between physical processor cores at run-time (if this is supported). For example, if a virtual core is linked to use FLASH local to Cortex-R core cluster 0, the virtual core would run more slowly if migrated to cluster 1.

Consider the Interrupt Latency and Real-time Requirements

A summary of the techniques applicable to applications with different interrupt latency and real-time requirements are summarised in the table below. A more detailed discussion can be found in subsequent sub-sections.

TABLE 5

Application Type	Interrupt Virtualization	Virtual Processor Cores	Para-virtualization/ Trap-and-emulate	Example Applications
Ultra-Low Latency Hard Real-Time	No	No	Yes, if the application allows it	Powertrain, Braking
Low-Latency Hard Real-Time	Yes, dynamic scheduling may be needed	Yes, dynamic scheduling may be needed	Yes, if the application allows it	Other chassis
Latency Focussed Real-Time	Yes	Yes	Yes, if the application allows it	General Body functions (e.g. immobilizer, ambient lightning, etc.)
Best Effort	Yes	Yes	Yes	Convenient Body functions (e.g. road noise suppression)

Ultra-low Latency Hard Real-time Applications

Here we are considering applications that require very short and predicable interrupt latencies. In essence, we want the "bare metal" behaviour. In these cases, interrupt virtualization and hosting multiple virtual cores on a physical core is more challenging, because of the increased and less predictable interrupt latencies that result from using these techniques.

Virtualizing GIC access, either with para-virtualization or trap-and-emulate, will reduce performance and may or may not be appropriate. While the hypervisor is carrying out GIC access on behalf of guest software, interrupts will most likely be disabled. However, this interrupt blocking will happen at times that are under the control of the guest software (e.g., during initialization).

The same argument applies to using EL2 device drivers for sharing peripherals and creating virtual peripherals. The para-virtualization or trap-and-emulate required is relatively slow, but when it occurs it is under the control of the application.

Using memory local to a core or cluster is particularly advisable for this type of application.

Low-latency Hard Real-time Applications

Here we are considering applications that require short and predictable interrupt latencies, but interrupt latencies longer and less predictable than "bare metal" are acceptable.

Depending on the acceptable upper bound on interrupt latencies for these types of applications, interrupt virtualization and hosting multiple virtual cores per physical core may be viable if a suitable virtual core scheduling algorithm is used. A dynamic scheduling algorithm, such as reservation-based scheduling with deferrable servers, can lead to fairly short and fairly predictable interrupt latencies – albeit longer than "bare metal" latencies because of the need for virtual core context switches when the virtual core that handles an interrupt is not currently running.

The discussion on virtualizing GIC access and using EL2 device drivers in the section on ultra-low latency hard real-time applications is also applicable here.

Latency Focussed Real-time Applications

Here we are considering applications that do not need particularly short interrupt latencies, but do need bounded interrupt latencies.

Again, depending on the acceptable upper bound on interrupt latencies, for these types of applications interrupt virtualization and hosting multiple virtual cores per physical core may be viable. Whether or not a static or dynamic virtual core scheduling is used will depend on the acceptable upper bound for interrupt latencies. If the upper bound is quite long, then a static scheduling algorithm may work. For example, if the longest time between a virtual core being scheduled is shorter than the upper bound on interrupt latency. If the upper bound is shorter, then a dynamic scheduling algorithm would be needed.

The discussion on virtualizing GIC access and using EL2 device drivers in the section on ultra-low latency hard real-time applications is also applicable here.

Best-effort Applications

For best effort (non-real time) applications, multiplexing multiple virtual cores onto a single physical core and shared and virtual device support can allow a far more optimal use of microcontroller resources. In some domains, many applications perform functions that have no hard real-time constraints, and these applications often perform their function in response to a stimulus and are then quiescent until the stimulus occurs again. Such systems are amenable to hosting multiple applications on a single physical core.

If the system needs to handle asynchronous events with short latencies, then a dynamic virtual core scheduling algorithm may be needed. However, if there is no need to handle asynchronous events with short latencies, a simple static scheduling algorithm will have a lower run-time overhead.

Legacy Software

When integrating legacy applications, one wants to minimize changes to the software. If the applications do not have hard real-time constraints, then most of the above mechanisms can be used except for para-virtualization. Trap-and-emulate would allow legacy peripherals to be emulated using EL2 device drivers.

If a legacy application has ultra-low latency hard real-time requirements, then the only solution may be to give the application a complete Cortex-R cluster and just firewall the cluster from the rest of the microcontroller using an SMPU and peripheral protection.

Recommendations for Future Microcontrollers

Provide Fine Grained Assignment of Peripherals to VMs

The need for EL2 device drivers can be reduced if peripherals can be assigned to VMs at a fine grain. For example, it would be useful to be able to assign individual Controller Area Network (CAN) channels, or even individual General-Purpose Input/Output (GPIO) pins, to a VM. This reduces the need for para-virtualization or trap-and-emulate with the consequent improvement in performance. While full peripheral virtualization would be ideal (see below), a compromise would be to use para-virtualization/trap-and-emulate to carry out initialization and configuration of peripherals, but allow direct access for the data plane.

Provide as Many MPU Regions as Possible

Assigning memory or peripherals to VMs often uses a lot of SMPU and core MPU regions. The more (S)MPU regions available the better (this statement implies that silicon partners include the EL2 MPU in the design synthesis). It is also good if MPU region granularity is small. A small MPU region granularity makes it easier to assign peripherals to VMs.

Support Virtualization in Peripherals

Even better than fine-grain peripheral to VM assignment is full virtualization of peripherals. Here a peripheral would be configured by a hypervisor to present a separate "view" of itself to multiple VMs. Each of the VMs could then use the peripheral as if it has been assigned a unique instance of the peripheral.

Ensure that DMA is Virtualization Aware

When a VM uses a DMA transfer, or uses a peripheral that uses a DMA transfer, the DMA transfer must not allow the VM to read or write from memory addresses that would normally be prohibited by the SMPU or core MPUs. The ideal would be for the DMA module/channel to automatically inherit the identity of the VM that configured the DMA module/channel or of the VM that configured the peripheral that uses the DMA module/channel. The VM identifier would then be checked by at least the SMPU on a DMA transfer, and the DMA transfer blocked if the VM did not have permission to read or write the memory involved in the DMA transfer. To support such behaviour, the Armv8-R VMID should be distributed to peripherals and DMA controllers.

If this automatic inheritance of VM identity is not possible, it should be possible to programmatically assign, by a privileged software entity, DMA modules/identifiers to VMs, so that SMPU control of DMA transfers can be done.

Control of DMA by VM, rather than just the bus-manager, is important when a single physical core may be running multiple VMs.

Adopt Available Virtualization Standards to Ease Software Mobility

Virtualization is a relatively new topic for the microcontrollers typically used to run Classic AUTOSAR systems, and naturally microcontroller manufacturers are adding features to give themselves an advantage over their competitors. However, users of these microcontrollers would like to develop software with the confidence that they can move that software to a different microcontroller, if necessary. Therefore, the combination of the microcontroller and a hypervisor needs to provide the users of microcontrollers with a fairly standard set of features and methods of feature usage. To this end, where industry standards are available, they should be adopted.

Some work as already been done in this area for the Armv8-R architecture, and a good example can be found in the Arm “Device virtualization principles for real time systems” paper.

Summary/Conclusion

The evolution of EE-architecture, including zonal controllers, demands further solutions for real-time software integration. Classic AUTOSAR is a de-facto standard in the automotive real-time software world, but further integration options, such as for legacy software, are a must moving forward. The Armv8-R architecture with the EL2 separation option represents a good option to enable intelligent integration options. How to use this integration option is highly dependent on the application though, and dedicated application demands will define which way of integration is most suitable.

This paper presents a variety of different techniques that can be used to support integration of different types of applications. This should help the system designer to understand how to best use Cortex-R52+ cores to support application integration.

Glossary (in alphabetical order):

- ABI — Application Binary Interface
- ASIL — Automotive Safety Integrity Level
- BOM — Bill Of Material
- EE — Electrical/electronic
- ECU — Electronic Control Unit
- EL2 — Exception Level 2
- GIC — Generic Interrupt Controller
- ISR — Interrupt Service Routine
- MCU — Microcontroller
- MMU — Memory Management Unit
- NEON — Arm Neon is an advanced single instruction multiple data (SIMD) architecture extension for the Arm Cortex-A and Arm Cortex-R series of processors
- OBD — On Board Diagnostics
- PPI — Private Peripheral Interrupt
- SGI — Software Generated Interrupt
- SMPU — System Memory Protection Unit
- SIMD — Single Instruction Multiple Data
- SoC — System on Chip
- VM — Virtual Machine